Distributed Computing
https://doi.org/10.1007/s00446-024-00466-4

Liveness and latency of Byzantine state-machine replication

Manuel Bravo[1] · Gregory Chockler[2] · Alexey Gotsman[3]

Received: 16 March 2023 / Accepted: 17 March 2024
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2024

Abstract

Byzantine state-machine replication (SMR) ensures the consistency of replicated state in the presence of malicious replicas and lies at the heart of the modern blockchain technology. Byzantine SMR protocols often guarantee safety under all circumstances and liveness only under synchrony. However, guaranteeing liveness even under this assumption is nontrivial. So far we have lacked systematic ways of incorporating liveness mechanisms into Byzantine SMR protocols, which often led to subtle bugs. To close this gap, we introduce a modular framework to facilitate the design of provably live and efficient Byzantine SMR protocols. Our framework relies on a *view* abstraction generated by a special *SMR synchronizer* primitive to drive the agreement on command ordering. We present a simple formal specification of an SMR synchronizer and its bounded-space implementation under partial synchrony. We also apply our specification to prove liveness and analyze the latency of three Byzantine SMR protocols via a uniform methodology. In particular, one of these results yields what we believe is the first rigorous liveness proof for the algorithmic core of the seminal PBFT protocol.

1 Introduction

Byzantine state-machine replication (SMR) [54] ensures the consistency of replicated state even when some of the replicas are malicious. It lies at the heart of the modern blockchain technology and is closely related to the classical Byzantine consensus problem. Unfortunately, no deterministic protocol can guarantee both safety and liveness of Byzantine SMR when the network is asynchronous [34]. A common way to circumvent this impossibility while maintaining determinism is to guarantee safety under all circumstances and liveness only when the network is synchronous. This is formalized by the *partial synchrony* model [25, 33], which stipulates that after some unknown *Global Stabilization Time* (GST) the system becomes synchronous, with message delays bounded by an unknown constant δ and process clocks tracking real time. Before GST, however, messages can be lost or arbitrarily delayed, and clocks at different processes can drift apart without bound.

Historically, researchers have paid more attention to safety of Byzantine SMR protocols than their liveness. For example, while the seminal PBFT protocol came with a detailed safety proof [22, §A], the nontrivial mechanisms ensuring its liveness were only given a brief informal justification [24, §4.5.1], which did not cover their most critical properties. However, ensuring liveness under partial synchrony is far from trivial, as illustrated by the many liveness bugs found in existing protocols [2, 6, 10, 21, 43]. In particular, classical failure detectors and leader oracles [25, 35] are of little help: while they have been widely used under benign failures [39, 40, 49], their implementations under Byzantine failures are either impractical [45] or detect only restricted failure types [31, 41, 48]. As an alternative, a textbook by Cachin et al. [19] proposed a leader oracle-like abstraction that accepts hints from the application to identify potentially faulty processes. However, as we explain in §9, their specification of the abstraction is impossible to implement, and in fact, the consensus algorithm constructed using it in [19] also suffers from a liveness bug.

Recent work on ensuring liveness has departed from failure detectors and instead revisited the approach of the original DLS paper [33]. This exploits the common structure of Byzantine consensus and SMR protocols under partial synchrony: such protocols usually divide their execution into *views*, each with a designated leader process that coordinates the protocol execution. If the leader is faulty, the processes switch to another view with a different leader. To ensure liveness, an SMR protocol needs to spend sufficient time in views

✉ Alexey Gotsman
dev@null.com

1 Informal Systems, Madrid, Spain

2 University of Surrey, Guildford, UK

3 IMDEA Software Institute, Madrid, Spain

that are entered by all correct processes and where the leader correctly follows the protocol. The challenge of achieving such *view synchronization* is that, before GST, clocks can diverge and messages that could be used to synchronize processes can get lost or delayed; even after GST, Byzantine processes may try to disrupt attempts to bring everybody into the same view. *View synchronizers* [14, 16, 50, 51, 59] encapsulate mechanisms for dealing with this challenge, allowing them to be reused across protocols.

View synchronizers have been mostly explored in the context of (single-shot) Byzantine consensus. In this case a synchronizer can just switch processes through an infinite series of views, so that eventually there is a view with a correct leader that is long enough to reach a decision [16, 51]. However, using such a synchronizer for SMR results in suboptimal solutions. For example, one approach is to use the classical SMR construction where each command is decided using a separate black-box consensus instance [54], implemented using a view synchronizer. However, this would force the processes in every instance to iterate over the same sequence of potentially bad views until the one with a correct leader and sufficiently long duration could be reached. Another approach is to use one view of a consensus synchronizer to agree on one command [59]. However, in this case a decision on each command is delayed until the consensus synchronizer generates a new view, thus increasing latency. We discuss the drawbacks of these approaches in more detail in §9.

To minimize the overheads of view synchronization, instead of automatically switching processes through views based on a fixed schedule, implementations such as PBFT allow processes to stay in the same view for as long as they are happy with its performance. The processes can then reuse a single view to decide multiple commands, usually with the same leader. To be useful for such SMR protocols, a synchronizer needs to allow the processes to control when they want to switch views via a special `advance` call. We call such a primitive an *SMR synchronizer*, to distinguish it from the less flexible *consensus synchronizer* introduced above. This kind of synchronizers was first introduced in [50, 51], but only used as an intermediate module to implement a consensus synchronizer. This line of work did not investigate the usability of SMR synchronizers as a generic building block for Byzantine SMR protocols.

In this paper we show that SMR synchronizers can be *directly* exploited to construct efficient and provably live SMR protocols and develop a general blueprint that enables such constructions. In more detail:

- We propose a formal specification of an SMR synchronizer (§3), which is simpler and more general than prior proposals [50, 51]. It is also strictly stronger than the consensus synchronizer of [16], which can be obtained from the SMR synchronizer at no extra cost. Informally, our specification guarantees that *(a)* the system will move to a new view if enough correct processes call `advance`, and *(b)* all correct processes will enter the new view, provided that for long enough, no correct process that enters this view asks to leave it. These properties enable correct processes to iterate through views in search of a well-behaved leader, and to synchronize in a view they are happy with.
- We give an SMR synchronizer implementation and prove that it satisfies our specification (§3.1). Unlike prior implementations [51], ours tolerates message loss before GST while using only bounded space; in practice, this feature is essential to defend against denial-of-service attacks [22]. We also provide a precise latency analysis of our synchronizer, quantifying how quickly all correct processes enter the next view after enough of them call `advance`.
- We demonstrate the usability of our synchronizer specification by applying it to construct and prove the correctness of several SMR protocols. First, we prove the liveness of a variant of PBFT using an SMR synchronizer (§4–6): to the best of our knowledge, this is the first rigorous proof of liveness for PBFT's algorithmic core. The proof establishes a strong liveness guarantee that implies censorship-resistance: every command submitted by a correct process will be executed. The use of the synchronizer specification in the proof allows us to abstract from view synchronization mechanics and focus on protocol-specific reasoning. This reasoning is done using a reusable methodology based on showing that the use of timers in the SMR protocol and the synchronizer together establish properties similar to those of failure detectors. The methodology also handles the realistic ways in which protocols such as PBFT adapt their timeouts to the unknown message delay δ. We demonstrate the generality of our methodology by also applying it to a version of PBFT with periodic leader changes [28, 57, 58] and a HotStuff-like protocol [59] (§8).
- We exploit the latency bounds for our synchronizer to establish both bad-case and good-case bounds for variants of PBFT implemented on top of it (§7). Our bad-case bound assumes that the protocol starts before GST; it shows that after GST all correct processes synchronize in the same view within a bounded time. This time is proportional to a conservatively chosen constant Δ that bounds post-GST message delays in all executions [42, 52]. Our good-case bound quantifies decision latency when the protocol starts after GST and matches the lower bound of [3].

2 System model

We consider a system of $n = 3f + 1$ processes. At most f of these can be Byzantine (aka *faulty*), i.e., can behave arbitrarily. The rest of the processes are *correct* and we denote their set by $\mathcal{C}$. We call a set Q of $2f + 1$ processes a *quorum* and write $\mathsf{quorum}(Q)$ in this case. We assume standard cryptographic primitives: processes can communicate via authenticated point-to-point links, sign messages using digital signatures, and use a collision-resistant hash function $\mathsf{hash}()$. We denote by $\langle m \rangle_i$ a message m signed by process p_i.

We consider a *partial synchrony* model [25, 33]: for each execution of the protocol, there exist a time GST and a duration δ such that after GST message delays between correct processes are bounded by δ. Before GST messages can get arbitrarily delayed or lost: this models the period when the network is unstable. As in [25], we assume that the values of GST and δ are unknown to the protocol. This reflects the requirements of practical systems, whose designers cannot accurately predict when network problems leading to asynchrony will stop and what the latency will be during the following synchronous period. We also assume that processes are equipped with hardware clocks that can drift unboundedly from real time before GST, but can accurately measure time thereafter.

3 SMR synchronizer specification and implementation

We consider a synchronizer interface defined in [50, 51], which here we call an *SMR synchronizer*. Let $\mathsf{View} = \{1, 2, \ldots\}$ be the set of *views*, ranged over by v; we use 0 to denote an invalid initial view. The synchronizer produces notifications `new_view`(v) at a process, telling it to *enter* view v. To trigger such view changes, the synchronizer allows a process to call a function `advance()`, which signals that the process wishes to *advance* to a higher view. We assume that a correct process does not call `advance` twice without an intervening `new_view` notification.

Our first contribution is the SMR synchronizer specification in Fig. 1, which is simpler and more general than prior proposals [50, 51] (see §9 for a discussion). The specification relies on the following notation. Given a view v entered by a correct process p_i, we denote by $E_i(v)$ the time when this happens; we let $E_{\text{first}}(v)$ and $E_{\text{last}}(v)$ denote respectively the earliest and the latest time when some correct process enters v. Similarly, we denote by $A_i(v)$ the time when a correct process p_i calls `advance` while in v, and let $A_{\text{first}}(v)$ and $A_{\text{last}}(v)$ denote respectively the earliest and the latest time when this happens. Given a partial function f, we write $f(x)\downarrow$ if $f(x)$ is defined, and $f(x)\uparrow$ if $f(x)$ is undefined. Thus, $E_{\text{first}}(v)\downarrow$ means that some correct process enters view v, and $E_{\text{first}}(v)\uparrow$ that no correct process enters this view.

The Monotonicity property in Fig. 1 ensures that views can only increase at a given process. Validity ensures that a process may only enter a view $v + 1$ if some correct process has called `advance` in v. This prevents faulty processes from disrupting the system by forcing view changes. As the following propositions shows, Validity implies that views are never skipped: no correct process can enter a view unless every preceding view has also been entered by some correct process. An important consequence of this property is that a common pattern in SMR protocols where leaders are rotated round-robin through views is guaranteed to find a correct leader. We use this fact in our proof of PBFT liveness (§6).

Proposition 1 (No skipped views)

$$\forall v, v'.\, 0 < v < v' \wedge E_{\text{first}}(v')\downarrow \implies E_{\text{first}}(v)\downarrow \wedge E_{\text{first}}(v) < E_{\text{first}}(v').$$

Proof Fix $v' \geq 2$ and assume that a correct process enters v', so that $E_{\text{first}}(v')\downarrow$. We prove by induction on k that

$$\forall k = 0..(v' - 1).\; E_{\text{first}}(v' - k)\downarrow \wedge E_{\text{first}}(v' - k) \leq E_{\text{first}}(v').$$

The base case of $k = 0$ is trivial. For the inductive step, assume that the required holds for some k. Then by Validity there exists a time $t < E_{\text{first}}(v' - k)$ at which some correct process p_j attempts to advance from $v' - k - 1$. But then p_j's view at t is $v' - k - 1$. Hence, p_j enters $v' - k - 1$ before t, so that

$$E_{\text{first}}(v' - k - 1) < t < E_{\text{first}}(v' - k) \leq E_{\text{first}}(v'),$$

as required. □

Bounded Entry ensures that, if some process enters view v, then all correct processes will do so within at most d time units of each other (our synchronizer satisfies this property for $d = 2\delta$). This only holds if within d no process attempts to advance to a higher view, as this may make some processes skip v and enter a higher view directly. Bounded Entry also holds only starting from some view $\mathcal{V}$, since a synchronizer may not be able to guarantee it for views entered before GST. When proving liveness of SMR protocols using the synchronizer, Bounded Entry helps us reason about the length of the interval during which all correct processes overlap in a given view $v \geq \mathcal{V}$. On the one hand, Bounded Entry guarantees that all processes enter v within at most d time units of each other; this can be used to ensure that the overlapping interval does not start too late. On the other hand, by Validity and Proposition 1, no correct process can enter a view $> v$ until

1. **Monotonicity.** A process enters increasing views:

$$\forall i, v, v'.\, E_i(v)\downarrow \wedge E_i(v')\downarrow \implies (v < v' \iff E_i(v) < E_i(v'))$$

2. **Validity.** A process only enters a view $v+1$ if some correct process has attempted to advance from v:

$$\forall i, v.\, E_i(v+1)\downarrow \implies A_{\text{first}}(v)\downarrow \wedge A_{\text{first}}(v) < E_i(v+1)$$

3. **Bounded Entry.** For some $\mathcal{V}$ and d, if a process enters a view $v \geq \mathcal{V}$ and no process attempts to advance to a higher view within time d, then all correct processes will enter v within d:

$$\exists \mathcal{V}, d.\, \forall v \geq \mathcal{V}.\, E_{\text{first}}(v)\downarrow \wedge \neg(A_{\text{first}}(v) < E_{\text{first}}(v) + d) \implies (\forall p_i \in \mathcal{C}.\, E_i(v)\downarrow) \wedge E_{\text{last}}(v) \leq E_{\text{first}}(v) + d$$

4. **Startup.** Some correct process will enter view 1 if $f+1$ processes call `advance`:

$$(\exists P \subseteq \mathcal{C}.\, |P| = f+1 \wedge (\forall p_i \in P.\, A_i(0)\downarrow)) \implies E_{\text{first}}(1)\downarrow$$

5. **Progress.** If a correct process enters a view v and, for some set P of $f+1$ correct processes, any process in P that enters v eventually calls `advance`, then some correct process will enter view $v+1$:

$$\forall v.\, E_{\text{first}}(v)\downarrow \wedge (\exists P \subseteq \mathcal{C}.\, |P| = f+1 \wedge (\forall p_i \in P.\, E_i(v)\downarrow \implies A_i(v)\downarrow)) \implies E_{\text{first}}(v+1)\downarrow$$

Fig. 1 SMR synchronizer specification

```
1 when the process starts or timer expires
2     advance();
3 upon new_view(v)
4     stop_timer(timer);
5     start_timer(timer, τ);
```

Fig. 2 A simple client of the SMR synchronizer

some correct process calls `advance` in v; this can be used to ensure that the interval does not end too soon.

The properties we have introduced so far ensure that all correct processes can synchronize in every view $\geq \mathcal{V}$ entered by some correct process. But they do not give any guarantees as to whether a correct process will be able to advance from a view $< \mathcal{V}$ or one with a faulty leader. The next two properties close this gap by establishing precise conditions under which a view change is guaranteed to occur.

Startup ensures that some correct process enters view 1 if $f+1$ processes call `advance`. Given a view v entered by a correct process, Progress determines conditions under which some correct process will enter the next view $v+1$. This will happen if for some set P of $f+1$ correct processes, any process in P entering v eventually calls `advance`. Note that even a single `advance` call at a correct process *may* lead to a view switch (reflecting the fact that in implementations faulty processes may help this correct process). Startup and Progress ensure that the synchronizer *must* switch if at least $f+1$ correct processes ask for this. We now illustrate a typical pattern of their use, which we later apply to PBFT (§6). To this end, we consider a simple client in Fig. 2, where in each view a process sets a timer for a fixed duration τ and calls `advance` when the timer expires. Using Startup and Progress we prove that this client keeps switching views forever as follows.

Proposition 2 *In any execution of the client in Fig. 2:* $\forall v.\, \exists v'.\, v' > v \wedge E_{\text{first}}(v')\downarrow$.

Proof Since all correct processes initially call `advance`, by Startup some correct process eventually enters view 1. Assume now that the proposition is false, so that there is a maximal view v entered by any correct process. Let P be any set of $f+1$ correct processes and consider an arbitrary process $p_i \in P$ that enters v. When this happens, p_i sets the timer for the duration τ. The process then either calls `advance` when timer expires, or enters a new view v' before this. In the latter case $v' > v$ by Monotonicity, which is impossible. Hence, p_i eventually calls `advance` while in v. Since p_i was chosen arbitrarily, $\forall p_i \in P.\, E_i(v)\downarrow \implies A_i(v)\downarrow$. Then by Progress we get $E_{\text{first}}(v+1)\downarrow$: a contradiction. □

As we illustrate in §6, the synchronizer properties work in tandem to ensure the liveness of an SMR protocol. Informally, Startup and Progress allow the processes to iterate over views in search for one with a well-behaved leader; Bounded Entry enables all correct processes to promptly enter that view; and Validity ensures that all correct processes can stay in it indefinitely despite any disruption by faulty processes.

A different type of a view synchronizer than the one we just presented has been used for Byzantine consensus [16, 51]—here we call it a *consensus synchronizer* (§1). Like an SMR synchronizer, a consensus synchronizer produces notifications telling a process to enter a new view v; we denote these by `new_consensus_view`(v) to distinguish them from `new_view` notifications of an SMR synchronizer. Unlike an SMR synchronizer, a consensus synchronizer lacks an `advance` call. Instead of letting the client protocol initiate view changes, this synchronizer keeps invoking `new_consensus_view` at processes at increasing intervals so that eventually the there is a view long enough for the consensus protocol running on top to decide. In §A

```
1  function advance()
2  |  send WISH(max(view + 1, view⁺)) to all;
3  |  advanced ← TRUE;

4  periodically                                   ▷ every ρ time units
5  |  if advanced then
6  |  |  send WISH(max(view + 1, view⁺)) to all;
7  |  else if view⁺ > 0 then
8  |  |  send WISH(view⁺) to all;

9  when received WISH(v) from p_j
10 |  prev_v, prev_v⁺ ← view, view⁺;
11 |  if v > max_views[j] then max_views[j] ← v;
12 |  view  ← max{v | ∃k. max_views[k] = v ∧
   |             |{j | max_views[j] ≥ v}| ≥ 2f + 1};
13 |  view⁺ ← max{v | ∃k. max_views[k] = v ∧
   |             |{j | max_views[j] ≥ v}| ≥ f + 1};
14 |  if view⁺ = view ∧ view > prev_v then
15 |  |  trigger new_view(view);
16 |  |  advanced ← FALSE;
17 |  if view⁺ > prev_v⁺ then
18 |  |  send WISH(view⁺) to all
```

Fig. 3 A bounded-space SMR synchronizer. Counters are initially 0

we show that we can use an SMR synchronizer to implement a consensus synchronizer without extra overhead, thereby demonstrating the generality of the former abstraction.

3.1 A bounded-space SMR synchronizer

In Fig. 3 we present a bounded-space algorithm that implements the specification in Fig. 1 under partial synchrony for $d = 2\delta$. Our implementation reuses algorithmic techniques from a consensus synchronizer we previously proposed [16] while supporting a more general abstraction. It thus inherits the key feature of the previous protocol: despite tolerating message loss before GST, the synchronizer only requires bounded space.

When a process calls advance (line 1), the synchronizer does not immediately move to the next view v', but disseminates a WISH(v') message announcing its intention. A process enters a new view once it accumulates a sufficient number of WISH messages supporting this. To achieve bounded space complexity, a process only keeps track of the highest WISH received from each peer. This information is stored in an array max_views : $\{1, \ldots, n\} \rightarrow$ View $\cup \{0\}$, whose jth entry tracks the maximal view received in a WISH message from process p_j (initially 0, updated in line 11). The content of max_views is then used to determine the views the process should enter. Since the process does not store each WISH it receives, we cannot require it to enter a new view upon receiving a set of matching WISHes for this exact view (as done, e.g., in Bracha broadcast [13, 50]): some of these WISHes may have been overwritten by higher ones in the max_views array. Instead, our implementation maintains two variables view and view$^+$, which respectively hold the $(2f+1)$st highest and the $(f+1)$st highest views in max_views (lines 12–13). These variables never decrease and always satisfy view $\leq$ view$^+$.

The process enters the view stored in the view variable when this variable increases (line 14; we explain the extra condition later). The process thus enters a view v only if it receives a quorum of WISHes for views $\geq v$. At least one of these WISHes must come from a correct process, which helps satisfy Validity. Note that a process may be forced to switch views even if it did not call advance. This helps lagging processes to catch up, but poses another challenge. Byzantine processes may equivocate, sending WISH messages to some processes but not others. In particular, they may send WISHes for views $\geq v$ to some correct process, helping it to form a quorum of WISHes sufficient for entering v. But they may withhold the same WISHes from another correct process, so that it fails to form a quorum for entering v, as necessary, e.g., for Bounded Entry. To deal with this, when a process receives a WISH that makes its view$^+$ increase, the process sends WISH(view$^+$) (line 18). The WISH(view$^+$) message replaces those that may have been omitted by Byzantine processes and helps other correct processes to quickly form the quorums of WISHes necessary to satisfy Bounded Entry. The mechanism of relaying WISH(view$^+$) also helps validate Progress: after GST, $f+1$ calls to advance by correct processes make the other correct processes also send WISHes, which helps correct processes accumulate a quorum of WISHes necessary to enter a new view.

The guard view$^+$ = view in line 14 ensures that a process does not enter a "stale" view such that another correct process already wishes to enter a higher one. Similarly, when the process calls advance, it sends a WISH for the maximum of view + 1 and view$^+$ (line 2). Thus, if view = view$^+$, so that the values of the two variables have not changed since the process entered the current view, then the process sends a WISH for the next view (view + 1). Otherwise, view < view$^+$, and the process sends a WISH for the higher view view$^+$.

Finally, to deal with message loss before GST, a process retransmits the highest WISH it sent every ρ time units, according to its local clock (line 4). Depending on whether the process has called advance in the current view (tracked by the advanced flag), the WISH is computed as in lines 18 or 2.

Space requirements Keeping track of only the maximal WISHes received in max_views allows our synchronizer to run in bounded space: it requires only $O(n)$ variables for storing views, despite tolerating message loss before GST. Furthermore, Proposition 1 ensures that views entered by correct processes throughout an execution do not skip values. Thus, although the individual view values stored by the synchronizer are unbounded, Proposition 1 limits the power of the adversary to exhaust their allocated space (similarly to [9]).

In practice, bounded space requirements of our protocol are essential to defend against denial-of-service attacks [22]: they prevent the attacker from crashing a process by flooding it with messages that cause it to run out of memory.

This stands in contrast to a naive synchronizer design that follows the approach of Bracha broadcast [13, 50]: enter a view v' upon receiving $2f+1$ WISH(v') messages, and echo WISH(v') upon receiving $f+1$ copies thereof. In this case a process would have to track all newly proposed views for which $< 2f+1$ WISHes have been received. Since messages sent before GST can be lost or delayed, this would require unbounded space.

Correctness The following theorem (proved in §B) states the correctness of our synchronizer.

Theorem 1 *Consider an execution with an eventual message delay δ. In this execution, the algorithm in Fig. 3 satisfies the properties in Fig. 1 for $d = 2\delta$ and*

$$\mathcal{V} = \max\{v \mid (E_{\text{first}}(v)\downarrow \wedge E_{\text{first}}(v) < \mathsf{GST} + \rho) \vee v = 0\} + 1,$$

if $A_{\text{first}}(0) < \mathsf{GST}$, and $\mathcal{V} = 1$, otherwise.

The theorem also gives witnesses for $\mathcal{V}$ and d in Bounded Entry, which is useful for analyzing the latency of SMR protocols using our synchronizer (§7). In particular, $\mathcal{V}$ is the next view after the highest one entered by a correct process at or before $\mathsf{GST} + \rho$ (or 1 if no view was entered).

4 PBFT using an SMR synchronizer

We now demonstrate how an SMR synchronizer can be used to implement Byzantine SMR. More formally, we implement *Byzantine atomic broadcast* [19], from which SMR can be implemented in the standard way [54]. This abstraction allows processes to broadcast *values*, and we assume an application-specific predicate to indicate whether a value is valid [20] (e.g., a block in a blockchain is invalid if it lacks correct signatures authorizing its transactions). We assume that all values broadcast by correct processes in a single execution are valid and unique. Then Byzantine atomic broadcast is defined by the following properties:

- **Integrity** Every process delivers a value at most once.
- **External Validity** A correct process delivers only values satisfying valid().
- **Ordering** If a correct process p delivers x_1 before x_2, then another correct process q cannot deliver x_2 before x_1.
- **Liveness** If a correct process broadcasts or delivers x, then eventually all correct processes will deliver x.

Note that Liveness implies *censorship-resistance*: the service cannot selectively omit values submitted by correct processes.

4.1 The PBFT-light protocol

We show how to implement Byzantine atomic broadcast via an SMR synchronizer using the example of a *PBFT-light* protocol (Figs. 4 and 5), which faithfully captures the algorithmic core of the Practical Byzantine Fault Tolerance protocol (PBFT) [23]. Whereas PBFT integrated view synchronization functionality with the core SMR protocol, PBFT-light delegates this functionality to an SMR synchronizer, and in §6 we rigorously prove its liveness when using any synchronizer satisfying our specification.

We base PBFT-light on the PBFT protocol with signatures and, for simplicity, omit the mechanisms for managing checkpoints and watermarks (hence the word “light”); these can be easily added without affecting liveness. The protocol works in a succession of views produced by the synchronizer. A process stores its current view in a variable curr_view. Each view v has a fixed *leader* $\mathsf{leader}(v) = p_{((v-1) \bmod n)+1}$ that is responsible for totally ordering values submitted for broadcast; the other processes are *followers*, which vote on proposals made by the leader. Processes store the sequence of values proposed by the leader in a log array; at the leader, a next counter points to the first free slot in the array. The protocol ensures that all values in the log array are unique. Processes monitor the leader’s behavior and ask the synchronizer to advance to another view if they suspect that the leader is faulty. A status variable records whether the process is operating as normal in the current view (NORMAL) or is changing the view.

4.2 Normal protocol operation

A process broadcasts a valid value x using a broadcast function (line 9). This keeps sending the value to all processes in a BROADCAST message until the process delivers the value. The periodic retransmission ensures that the value

```
1  function start()
2  | if curr_view = 0 then advance();

3  when a timer expires
4  | stop_all_timers();
5  | advance();
6  | status ← ADVANCED;
7  | dur_delivery ← dur_delivery + τ;
8  | dur_recovery ← dur_recovery + τ;

9  function broadcast(x)
10 | pre: valid(x);
11 | send ⟨BROADCAST(x)⟩_i to all periodically
   |   until x is delivered

12 when received BROADCAST(x)
13 | pre: valid(x) ∧ status = NORMAL ∧
   |        (timer_delivery[x] not active) ∧
   |        (∀k. k ≤ last_delivered ⟹ commit_log[k] ≠ x);
14 | start_timer(timer_delivery[x], dur_delivery);
15 | send ⟨FORWARD(x)⟩_i to leader(curr_view);

16 when received FORWARD(x)
17 | pre: valid(x) ∧ status = NORMAL ∧
   |        p_i = leader(curr_view) ∧ ∀k. log[k] ≠ x;
18 | send ⟨PREPREPARE(curr_view, next, x)⟩_i to all;
19 | next ← next + 1;

20 when received ⟨PREPREPARE(v, k, x)⟩_j
21 | pre: p_j = leader(v) ∧ curr_view = v ∧
   |        status = NORMAL ∧ phase[k] = START ∧
   |        valid(x) ∧ (∀k'. log[k'] ≠ x)
22 | (log, phase)[k] ← (x, PREPREPARED);
23 | send ⟨PREPARE(v, k, hash(x))⟩_i to all;

24 when received {⟨PREPARE(v, k, h)⟩_j | p_j ∈ Q} = C
     for a quorum Q
25 | pre: curr_view = v ∧
   |        phase[k] = PREPREPARED ∧
   |        status = NORMAL ∧ hash(log[k]) = h;
26 | (prep_log, prep_view, cert, phase)[k] ←
   |    (log[k], curr_view, C, PREPARED);
27 | send ⟨COMMIT(v, k, h)⟩_i to all;

28 when received {⟨COMMIT(v, k, h)⟩_j | p_j ∈ Q} = C
     for a quorum Q
29 | pre: curr_view = v ∧ phase[k] = PREPARED ∧
   |        status = NORMAL ∧ hash(prep_log[k]) = h;
30 | (commit_log, phase)[k] ← (log[k], COMMITTED);
31 | broadcast ⟨DECISION(commit_log[k], k, C)⟩;

32 when received DECISION(x, k, C)
33 | pre: commit_log[k] ≠ ⊥ ∧
   |        ∃v. committed(C, v, k, hash(x));
34 | commit_log[k] ← x;

35 when commit_log[last_delivered + 1] ≠ ⊥
36 | last_delivered ← last_delivered + 1;
37 | if commit_log[last_delivered] ≠ nop then
38 | | deliver(commit_log[last_delivered])
39 | stop_timer(
   |   timer_delivery[commit_log[last_delivered]]);
40 | if last_delivered = init_log_length ∧
   |  status = NORMAL then
41 | | stop_timer(timer_recovery);

42 upon new_view(v)
43 | stop_all_timers();
44 | curr_view ← v;
45 | status ← INITIALIZING;
46 | send ⟨NEW_LEADER(curr_view, prep_view,
   |  prep_log, cert)⟩_i to leader(curr_view);
47 | start_timer(timer_recovery, dur_recovery);

48 when received {⟨NEW_LEADER(v, prep_view_j,
     prep_log_j, cert_j)⟩_j | p_j ∈ Q} = M
     for a quorum Q
49 | pre: p_i = leader(v) ∧ curr_view = v ∧
   |        status = INITIALIZING ∧
   |        ∀m ∈ M. ValidNewLeader(m);
50 | forall k do
51 | | if ∃p_j' ∈ Q. prep_view_j'[k] ≠ 0 ∧
   | |  ∀p_j ∈ Q. prep_view_j[k] ≤ prep_view_j'[k]
   | |  then log'[k] ← prep_log_j'[k];
52 | next ← max{k | log'[k] ≠ ⊥};
53 | forall k = 1..(next − 1) do
54 | | if log'[k] = ⊥ ∨ ∃k'. k' ≠ k ∧
   | |  log'[k'] = log'[k] ∧ ∃p_j' ∈ Q. ∀p_j ∈ Q.
   | |  prep_view_j'[k'] > prep_view_j[k] then
55 | | | log'[k] ← nop
56 | send ⟨NEW_STATE(v, log', M)⟩_i to all;

57 when received ⟨NEW_STATE(v, log', M)⟩_j = m
58 | pre: status = INITIALIZING ∧
   |        curr_view = v ∧ ValidNewState(m);
59 | log ← log';
60 | forall {k | log[k] ≠ ⊥} do
61 | | phase[k] ← PREPREPARED;
62 | | send ⟨PREPARE(v, k, hash(log[k]))⟩_i to all;
63 | status ← NORMAL;
64 | init_log_length ← max{k | log[k] ≠ ⊥};
65 | if init_log_length ≤ last_delivered then
66 | | stop_timer(timer_recovery);
```

Fig. 4 The PBFT-light protocol at a process p_i. Auxiliary predicates are defined in Fig. 5

$$\mathsf{prepared}(C, v, k, h) \iff \exists Q.\, \mathsf{quorum}(Q) \land C = \{\langle \mathtt{PREPARE}(v, k, h)\rangle_j \mid p_j \in Q\}$$
$$\mathsf{committed}(C, v, k, h) \iff \exists Q.\, \mathsf{quorum}(Q) \land C = \{\langle \mathtt{COMMIT}(v, k, h)\rangle_j \mid p_j \in Q\}$$
$$\mathsf{ValidNewLeader}(\langle \mathtt{NEW_LEADER}(v, prep_view, prep_log, cert)\rangle_{_}) \iff$$
$$\forall k.\, (prep_view[k] > 0 \implies prep_view[k] < v \land \mathsf{prepared}(cert[k], prep_view[k], k, prep_log[k]))$$
$$\mathsf{ValidNewState}(\langle \mathtt{NEW_STATE}(v, log', M)\rangle_i) \iff p_i = \mathsf{leader}(v) \land \exists Q, prep_view, prep_log, cert.$$
$$\mathsf{quorum}(Q) \land M = \{\langle \mathtt{NEW_LEADER}(v, prep_view_j, log_j, cert_j)\rangle_j \mid p_j \in Q\} \land$$
$$(\forall m \in C.\, \mathsf{ValidNewLeader}(m)) \land (log' \text{ is computed from } M \text{ as per lines 50-55})$$

Fig. 5 Auxiliary predicates for PBFT-light

will reach the processes despite message loss before GST. When a process receives a BROADCAST message with a new value (line 12), it forwards the value to the leader in a FORWARD message. This ensures that the value reaches the leader even when broadcast by a faulty process, which may withhold the BROADCAST message from the leader. (We explain the timer set in line 14 later.) When the leader receives a new value x in a FORWARD message (line 16), it sends a PREPREPARE message to all processes (including itself) that includes x and its position in the log, generated from the next counter. Processes vote on the leader's proposal in two phases, each with an all-to-all message exchange. A process keeps track of the status of values going through the vote in an array phase, whose entries initially store START.

When a process receives a proposal x for a position k from the leader of its view v (line 20), it first checks that $\mathsf{phase}[k] = \text{START}$, so that it has not yet accepted a proposal for the position k in the current view. It also checks that the value is valid and distinct from all values it knows about. The process then stores x in $\mathsf{log}[k]$ and advances $\mathsf{phase}[k]$ to PREPREPARED. Since a faulty leader may send different proposals for the same position to different processes, the process next communicates with others to check that they received the same proposal. To this end, it disseminates a PREPARE message with the position and the hash of the value x it received. The process handles x further once it gathers a set C of PREPARE messages from a quorum matching the value (line 24), which we call a *prepared certificate* and check using the prepared predicate in Fig. 5. In this case the process stores the value in $\mathsf{prep_log}[k]$, the certificate in $\mathsf{cert}[k]$, and the view in which it was formed in $\mathsf{prep_view}[k]$. At this point we say that the process *prepared* the proposal, as recorded by setting its phase to PREPARED. As we now show, processes cannot prepare different values at the same position and view, given that each correct process can send only one corresponding PREPARE message. Formally, let us write $\mathsf{wf}(C)$ (for *well-formed*) if the set of correctly signed messages C were generated in the execution of the protocol.

Proposition 3

$$\forall k, v, C, C', x, x'.\, \mathsf{prepared}(C, v, k, \mathsf{hash}(x)) \land \mathsf{prepared}(C', v, k, \mathsf{hash}(x')) \land \mathsf{wf}(C) \land \mathsf{wf}(C') \implies x = x'.$$

Proof By contradiction, suppose that $x \neq x'$. Because a prepared certificate consists of at least $2f + 1$ PREPARE messages and there are $3f + 1$ processes in total, there must be a correct process that sent two PREPARE messages with different hashes for the same position and view. But this is impossible due to the check on the check on phase in line 21. □

Since the synchronizer transitions processes through increasing views (Monotonicity in Fig. 1), we also get:

Proposition 4 *The variables* curr_view *and* $\mathsf{prep_view}[k]$ *(for any k) at a correct process never decrease and we always have* $\mathsf{prep_view}[k] \leq \mathsf{curr_view}$.

Having prepared a value, the process participates in another message exchange: it disseminates a COMMIT message with its hash. Once the process gathers a quorum of matching COMMIT messages (line 28), it stores the value in a commit_log array and advances its phase to COMMITTED: we say that the value is now *committed*. The protocol ensures that correct processes cannot commit different values at the same position, even in different views. We call a quorum of matching COMMIT messages a *commit certificate* and check it using the committed predicate in Fig. 5. A process delivers committed values in the commit_log order, with last_delivered tracking the last delivered position.

To satisfy the Liveness property of atomic broadcast, similarly to [10], PBFT-light allows a process to find out about committed values from other processes directly. When a process commits a value (line 28), it disseminates a DECISION message with the value, its position k in the log and the commit certificate (line 31). A process receiving a DECISION with a valid certificate saves the value in $\mathsf{commit_log}[k]$, which allows it to be delivered (line 35). The DECISION messages are disseminated via reliable broadcast ensuring that, if one correct process delivers the value, then so do all others. To implement this, each process could periodically resend the DECISION messages it has (omitted from

the pseudocode). A more practical implementation would only resend information that other processes are missing. As proved in [33], such periodic resends are unavoidable in the presence of message loss.

4.3 View initialization

When the synchronizer tells a process to move to a new view v (line 42), the process sets curr_view to v, which ensures that it will no longer accept messages from prior views. It also sets status to INITIALIZING, which means that the process is not yet ready to order values in the new view. It then sends a `NEW_LEADER` message to the leader of v with the information about the values it has prepared so far and their certificates[1].

The new leader waits until it receives a quorum of well-formed `NEW_LEADER` messages, as checked by the predicate ValidNewLeader (line 48). Based on these, the leader computes the initial log of the new view, stored in log'. Similarly to Paxos [47], for each index k the leader puts at the kth position in log' the value prepared in the highest view (line 50). The resulting array may contain empty or duplicate entries. To resolve this, the leader writes `nop` into empty entries and those entries for which there is a duplicate prepared in a higher view (line 53). The latter is safe because one can show that no value could have been committed in such entries in prior views. Finally, the leader sends a `NEW_STATE` message to all processes, containing the initial log and the `NEW_LEADER` messages from which it was computed (line 58).

A process receiving the `NEW_STATE` message first checks its correctness by redoing the leader's computation, as specified by the ValidNewState predicate (line 57). If the check passes, the process overwrites its log with the new one and sets status to NORMAL. It also sends `PREPARE` messages for all log entries, to commit them in the new view. A more practical implementation would include a checkpointing mechanism, so that a process restarts committing previous log entries only from the last stable checkpoint [23]; this mechanism can be easily added to PBFT-light.

4.4 Triggering view changes

We now describe when a process calls `advance`, which is key to ensure liveness (§6). This happens either on start-up (line 2) or when the process suspects that the current leader is faulty. To this end, the process monitors the leader's behavior using timers; if one of these expires, the process calls `advance` and sets status to ADVANCED (line 3). First, the process checks that each value it receives is delivered promptly: e.g., to guard against a faulty leader not making any proposals or censoring certain values. For a value x this is done using timer_delivery$[x]$, set for a duration dur_delivery when the process receives `BROADCAST`(x) (lines 14). The timer is stopped when the process delivers x (line 39). A process also checks that the leader initializes a view quickly enough: e.g., to guard against the leader crashing during the initialization. Thus, when a process enters a view it starts timer_recovery for a duration dur_recovery (line 47). The process stops the timer when it delivers all values in the initial log (lines 41 and 66). The above checks may make a process suspect a correct leader if the timeouts are initially set too small with respect to the message delay δ, unknown to the process. To deal with this, a process increases dur_delivery and dur_recovery each time a timer expires, which signals that the current view is not operating normally (lines 7-8). This way of adapting timeouts is used in the original PBFT [22, §2.3.5].

4.5 Space requirements

Since the synchronizer is not guaranteed to switch processes between views all at the same time, a process in a view v may receive a message from a higher view $v' > v$, which needs to be stored in case the process finally switches to v'. If implemented naively, this would require a process to store unboundedly many messages. Instead, we allow a process to store, for each log position, message type and sender, only the message of this type received from this sender that has the highest view. Recall that our synchronizer from §3.1 uses only bounded space. Together with the above optimization, this reduces the amount of space required by the SMR protocol. The original PBFT further prevents Byzantine processes from exhausting the space of log positions [23], which allows bounding the overall space required by the protocol. However, for simplicity we omit watermarks from PBFT-light.

5 Safety of PBFT-light

We now prove the safety of the variant of PBFT we just presented. In PBFT-light, committing a value requires preparing it, which implies

Proposition 5

$$\forall k, v, C, h.\, \mathsf{committed}(C, v, k, h) \wedge \mathsf{wf}(C) \implies \exists C'.\, \mathsf{prepared}(C', v, k, h) \wedge \mathsf{wf}(C').$$

[1] In PBFT this information is sent in `VIEW-CHANGE` messages, which also play a role similar to `WISH` messages in our synchronizer (Fig. 3). In PBFT-light we opted to eschew `VIEW-CHANGE` messages to maintain a clear separation between view synchronization internals and the SMR protocol.

Furthermore, the validity checks in the protocol ensure that any prepared value is valid:

Proposition 6

$$\forall k, v, C, x.\, \mathsf{prepared}(C, v, k, \mathsf{hash}(x)) \wedge \mathsf{wf}(C) \Longrightarrow \mathsf{valid}(x).$$

The above two propositions imply

Corollary 1 *PBFT-light satisfies External Validity.*

We first prove the following auxiliary result, which states that logs sent in `NEW_STATE` messages never have duplicate entries apart from `nop`s.

Lemma 1 *If* $m = \langle \mathtt{NEW_STATE}(v', log', M)\rangle_{\mathsf{leader}(v')}$ *is a sent message such that* ValidNewState(m)*, then*

$$\forall k, k'.\, log'[k] = log'[k'] \notin \{\bot, \mathtt{nop}\} \Longrightarrow k = k'.$$

Proof We prove the statement of the lemma by induction on v'. Assume this holds for all $v' < v^*$; we now prove it for $v' = v^*$. Let a set M be defined as

$$\{\langle \mathtt{NEW_LEADER}(v', prep_view_j, log_j, cert_j)\rangle_j \mid p_j \in Q\}$$

for some quorum Q. By contradiction, assume that for some k, k' and x we have $k \neq k'$ and

$$log'[k] = log'[k'] = x \notin \{\bot, \mathtt{nop}\}.$$

Since $\mathsf{ValidNewState}(m)$, log' is computed from M as per lines 50-55. Then due to the loop at line 53, for some $i, i' \in Q$ we have $log_i[k] = log_{i'}[k'] = x$ and $prep_view_i[k] = prep_view_{i'}[k'] = v$ for some v such that $0 < v < v'$. Hence, for some C and C' we have

$$\mathsf{prepared}(C, v, k, \mathsf{hash}(x)) \wedge \mathsf{prepared}(C', v, k', \mathsf{hash}(x)) \wedge \mathsf{wf}(C) \wedge \mathsf{wf}(C').$$

Because a prepared certificate consists of at least $2f + 1$ `PREPARE` messages and there are $3f + 1$ processes in total, there must be a correct process that sent messages $\mathtt{PREPARE}(v, k, \mathsf{hash}(x))$ and $\mathtt{PREPARE}(v, k', \mathsf{hash}(x))$. But this is impossible because by the induction hypothesis, the process starts the view v with a log without duplications (except `nop`s), and does not add duplicate entries due to the check at line 21. This contradiction demonstrates the required. □

The above lemma implies that a value different from `nop` cannot be prepared at different positions in the same view.

Corollary 2

$$\forall x, v, k, k', C, C'.\, \mathsf{prepared}(C, v, k, \mathsf{hash}(x)) \wedge \mathsf{prepared}(C', v, k', \mathsf{hash}(x)) \wedge \mathsf{wf}(C) \wedge \mathsf{wf}(C') \wedge x \neq \mathtt{nop} \Longrightarrow k = k'.$$

Proof Assume the contrary. Because a prepared certificate consists of at least $2f + 1$ `PREPARE` messages and there are $3f + 1$ processes in total, there must be a correct process that sent messages $\mathtt{PREPARE}(v, k, \mathsf{hash}(x))$ and $\mathtt{PREPARE}(v, k', \mathsf{hash}(x))$. But this is impossible because by Lemma 1, the process starts the view v with a log without duplications (except `nop`s), and does not add duplicate entries due to the check at line 21. This contradiction demonstrates the required. □

The Ordering and Integrity properties of PBFT-light follow from the next lemma, which assumes that a committed certificate was assembled for a value x at a position k in a given view v. The lemma ensures that in this case: *(i)* only x can be prepared at the position k in any view higher than v; and *(ii)* x cannot be prepared at a position different from k in any view higher than v, unless x is a dummy `nop` value.

Lemma 2 *Fix* k, v, v', C *and* x*, and assume*

$$\mathsf{committed}(C, v, k, \mathsf{hash}(x)) \wedge \mathsf{wf}(C) \wedge v' > v.$$

Then:

- $\forall C', x'.\, \mathsf{prepared}(C', v', k, \mathsf{hash}(x')) \wedge \mathsf{wf}(C') \Longrightarrow x = x'$.
- $\forall C', k'.\, x \neq \mathtt{nop} \wedge \mathsf{prepared}(C', v', k', \mathsf{hash}(x)) \wedge \mathsf{wf}(C') \Longrightarrow k = k'$.

Proof We prove the statement of the lemma by induction on v'. Assume this holds for all $v' < v^*$; we now prove it for $v' = v^*$. Thus, we have

$$\forall C'', k'', v''.\, v < v'' < v' \wedge x \neq \mathtt{nop} \wedge \mathsf{prepared}(C'', v'', k'', \mathsf{hash}(x)) \wedge \mathsf{wf}(C'') \Longrightarrow k = k''. \tag{1}$$

The induction hypothesis also implies

$$\forall C'', x'', v''.\, v < v'' < v' \wedge \mathsf{prepared}(C'', v'', k, \mathsf{hash}(x'')) \wedge \mathsf{wf}(C'') \Longrightarrow x = x''.$$

Furthermore, by Propositions 3 and 5 we have

$$\forall C'', x''.\, \mathsf{prepared}(C'', v, k, \mathsf{hash}(x'')) \wedge \mathsf{wf}(C'') \Longrightarrow x = x'',$$

so that overall we get

$$\forall C'', x'', v''. v \le v'' < v' \wedge \mathsf{prepared}(C'', v'', k, \mathsf{hash}(x'')) \wedge \mathsf{wf}(C'') \implies x = x''. \quad (2)$$

Assume now that $\mathsf{prepared}(C', v', k, \mathsf{hash}(x'))$ and $\mathsf{wf}(C')$. Then a correct process that sent the corresponding $\texttt{PREPARE}$ message must have received a message $m = \texttt{NEW_STATE}(v', log', M)$ from the leader of v' satisfying $\mathsf{ValidNewState}(m)$. Let a set M be defined as

$$\{\langle \texttt{NEW_LEADER}(v', prep_view_j, log_j, cert_j)\rangle_j \mid p_j \in Q\}$$

for some quorum Q. Since $\mathsf{ValidNewState}(m)$, we have $\forall m' \in M. \mathsf{ValidNewLeader}(m')$, so that

$$\forall p_j \in Q. prep_view_j < v' \wedge (prep_view_j \neq 0 \implies \mathsf{prepared}(cert_j, prep_view_j, k, \mathsf{hash}(log_j[k])) \wedge \mathsf{wf}(cert_j)).$$

From this and (2) we get that

$$\forall p_j \in Q. prep_view_j \ge v \implies log_j[k] = x. \quad (3)$$

Since $\mathsf{committed}(C, v, k, \mathsf{hash}(x))$, a quorum Q' of processes sent $\texttt{COMMIT}(v, k, \mathsf{hash}(x))$. The quorums Q and Q' have to intersect in some correct process p_i, which has thus sent both $\texttt{COMMIT}(v, k, \mathsf{hash}(x))$ and $\texttt{NEW_LEADER}(v', prep_view_i, log_i, cert_i)$. Since $v < v'$, this process p_i must have sent the $\texttt{COMMIT}$ message before the $\texttt{NEW_LEADER}$ message. Before sending $\texttt{COMMIT}(v, k, \mathsf{hash}(x))$ the process set $\mathsf{prep_view}[k]$ to v (line 26). Then by Proposition 4 process p_i must have had $\mathsf{prep_view}[k] \ge v$ when it sent the $\texttt{NEW_LEADER}$ message. Hence, $prep_view_i[k] \ge v > 0$ and $\max\{prep_view_{j'}[k] \mid p_{j'} \in Q\} \ge v$. Then from (3) we get

$$\forall p_j \in Q. prep_view_j[k] = \max\{prep_view_{j'}[k] \mid p_{j'} \in Q\} \implies log_j[k] = x. \quad (4)$$

Assume now that $x \neq \texttt{nop}$, but $log'[k] = \texttt{nop}$ due to line 55. Then

$$\exists k'. k' \neq k \wedge log'[k'] = x \wedge \exists p_j \in Q. \forall p_{j'} \in Q. prep_view_j[k'] > prep_view_{j'}[k]$$

and

$$v' > prep_view_j[k'] > prep_view_i[k] \ge v.$$

Since $\mathsf{ValidNewState}(m)$, for some C'' we have $\mathsf{prepared}(C'', prep_view_j[k'], k', \mathsf{hash}(x))$ and $\mathsf{wf}(C'')$. Then by (1) we have $k = k'$, which yields a contradiction. This together with (4) and $\mathsf{ValidNewState}(m)$ implies $log'[k] = x$, as required.

Assume now $x \neq \texttt{nop}$, $\mathsf{prepared}(C', v', k', \mathsf{hash}(x))$ and $\mathsf{wf}(C')$. Then a correct process that sent the corresponding $\texttt{PREPARE}$ message must have received a message $m = \texttt{NEW_STATE}(v', log', M)$ from the leader of v' satisfying $\mathsf{ValidNewState}(m)$. As before, we can show $log'[k] = x$. By Lemma 1, the process starts the view v' with a log without duplications (except $\texttt{nop}$s), and does not add duplicate entries due to the check at line 21. Hence, we must have $k' = k$, as required. □

Corollary 3 *PBFT-light satisfies Ordering.*

Proof By contradiction, assume that Ordering is violated. Then for some k, two correct processes execute the handler in line 35 for $\mathsf{last_delivered} = k - 1$ so that $\mathsf{commit_log}[k] = x$ at one process and $\mathsf{commit_log}[k] = x'$ at the other, where $x \neq x'$. Then

$$\mathsf{committed}(C, v, k, \mathsf{hash}(x)) \wedge \mathsf{committed}(C', v', k, \mathsf{hash}(x'))$$

for some well-formed C and C'. By Proposition 5 we have

$$\mathsf{prepared}(C_0, v, k, \mathsf{hash}(x)) \wedge \mathsf{prepared}(C'_0, v', k, \mathsf{hash}(x'))$$

for some well-formed C_0 and C'_0. Without loss of generality assume $v \le v'$. If $v = v'$, then $x = x'$ by Proposition 3. If $v < v'$, then $x = x'$ by Lemma 2. In either case we get a contradiction. □

Corollary 4 *PBFT-light satisfies Integrity.*

Proof By contradiction, assume that Integrity is violated. Then for some k, k' such that $k \neq k'$ and $x \neq \texttt{nop}$, a correct process executes the handler in line 35 first in a view v for $\mathsf{last_delivered} = k-1$ and $\mathsf{commit_log}[k] = x$ and then in a view v' for $\mathsf{last_delivered} = k'-1$ and $\mathsf{commit_log}[k'] = x$. We must have

$$\mathsf{committed}(C, v, k, \mathsf{hash}(x)) \wedge \mathsf{committed}(C', v', k', \mathsf{hash}(x'))$$

for some well-formed C and C'. By Proposition 5 we have

$$\mathsf{prepared}(C_0, v, k, \mathsf{hash}(x)) \wedge \mathsf{prepared}(C'_0, v', k', \mathsf{hash}(x'))$$

for some well-formed C_0 and C'_0. Without loss of generality assume $v \leq v'$. If $v = v'$, then we get a contradiction by Corollary 2. If $v < v'$, then we get a contradiction by Lemma 2. □

6 Liveness of PBFT-light

Assume that PBFT-light is used with a synchronizer satisfying the specification in Fig. 1; to simplify the following latency analysis we let $d = 2\delta$, as for the synchronizer in Fig. 3. We now prove that PBFT-light satisfies the Liveness property of Byzantine atomic broadcast; together with the safety proof in §5, this establishes the correctness of the protocol. To the best of our knowledge, this is the first rigorous proof of liveness for the algorithmic core of PBFT: as we elaborate in §9, the liveness mechanisms of PBFT came only with a brief informal justification, which did not cover their most critical properties [24, §4.5.1]. Our proof is simplified by the use of the synchronizer specification, which allows us to abstract from view synchronization mechanics and focus on protocol-specific reasoning.

We prove the liveness of PBFT-light by showing that the protocol establishes properties reminiscent of those of failure detectors [25]. First, similarly to their completeness property, we prove that every correct process eventually attempts to advance from a *bad* view in which no progress is possible (e.g., because the leader is faulty). Intuitively, this holds because in PBFT-light each process monitors the leader's behavior using timers.

Lemma 3 *Assume that a correct process p_i receives* BROADCAST(x) *for a valid value x while in a view v. If p_i never delivers x and never enters a view higher than v, then it eventually calls* advance *in v.*

Proof We know that at some point p_i enters view v, and at this moment it starts timer_recovery. If the timer expires, then p_i calls advance in v, as required. Assume that timer_recovery does not expire at p_i. Then p_i stops the timer at lines 4, 41, 66 or 43. The latter is impossible, as this would imply that p_i enters a higher view. If p_i stops the timer at line 4, then it calls advance in v, as required. Assume now that p_i stops the timer at lines 41 or 66. This implies that p_i sets status = NORMAL at some point while in v. If p_i sets status = ADVANCED while in v, then it calls advance in v, as required. Thus, it remains to consider the case when p_i sets status = NORMAL at some point while in v and does not change it while in this view. Since p_i receives BROADCAST(x) for a valid value x while in a view v, the handler at line 12 is executed at some point. At this point p_i starts timer_delivery[x]. If the timer expires, then p_i calls advance in v, as required. Otherwise p_i stops the timer at lines 4, 39 or 43. The last two are impossible, as this would imply that p_i enters a higher view or that x is delivered. In the remaining case p_i calls advance in v, as required. □

Note that even though the above lemma ensures that correct processes will eventually try to advance from a view with a faulty leader, the SMR protocol by itself does not ensure that the processes will actually enter a new view with a different leader. Achieving this is the job of the synchronizer, and in the following proof of PBFT-light we show that this happens using the Progress property in Fig. 1 together with Lemma 3.

Our next lemma is similar to the eventual accuracy property of failure detectors. It stipulates that if the timeout values are high enough, then eventually any correct process that enters a *good* view (with a correct leader) will never attempt to advance from it. The Validity property of the synchronizer then ensures that no correct process will leave the good view. Let $\mathsf{dur_recovery}_i(v)$ and $\mathsf{dur_delivery}_i(v)$ denote respectively the value of dur_recovery and dur_delivery at a correct process p_i while in view v.

Lemma 4 *Consider a view $v \geq \mathcal{V}$ such that $E_{\mathrm{first}}(v) \geq \mathsf{GST}$ and $\mathsf{leader}(v)$ is correct. If $\mathsf{dur_recovery}_i(v) > 6\delta$ and $\mathsf{dur_delivery}_i(v) > 4\delta$ at each correct process p_i that enters v, then no correct process calls* advance *in v.*

Before proving the lemma, we informally explain the rationale for the bounds on timeouts in it, using the example of dur_recovery. The timer timer_recovery is started at a process p_i when this process enters a view v (line 47), and is stopped when the process delivers all values inherited from previous views (lines 41 or 66). The two events are separated by 4 communication steps of PBFT-light, exchanging messages of the types NEW_LEADER, NEW_STATE, PREPARE and COMMIT (Fig. 6). However, 4δ would be too small a value for dur_recovery. This is because the leader of v sends its NEW_STATE message only after receiving a quorum of NEW_LEADER messages, and different processes may enter v and send their NEW_LEADER messages at different times (e.g., p_i and p_j in Fig. 6). Hence, dur_recovery must additionally accommodate the maximum discrepancy in the entry times, which is $d = 2\delta$ by the Bounded Entry property. Then to ensure that p_i stops the timer before it expires, we require $\mathsf{dur_recovery}_i(v) > 6\delta$. As the above reasoning illustrates, Lemma 4 is more subtle than Lemma 3: while the latter is ensured just by the checks in the SMR protocol, the former relies on the Bounded Entry property of the synchronizer.

Another subtlety about Lemma 4 is that the δ used in its premise is a priori unknown. Hence, to apply the lemma in the liveness proof of PBFT-light, we have to argue that, if correct processes keep changing views due to lack of progress, then all of them will eventually increase their timeouts high enough to satisfy the bounds in Lemma 4. This is nontrivial due to the fact that, as in the original PBFT [22, §2.3.5], in our

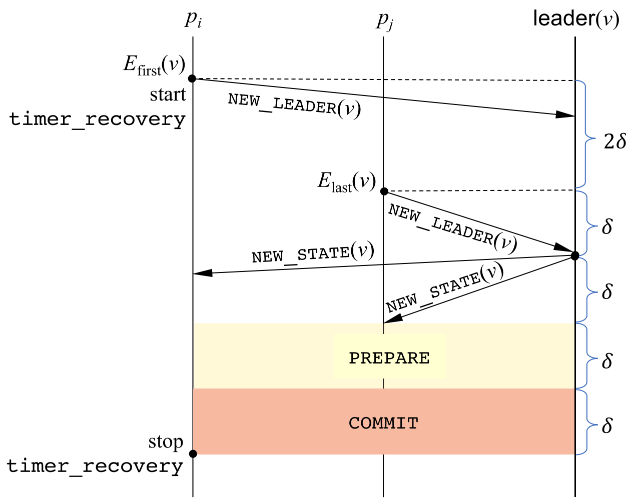

Fig. 6 An illustration of the bound on dur_recovery in Lemma 4

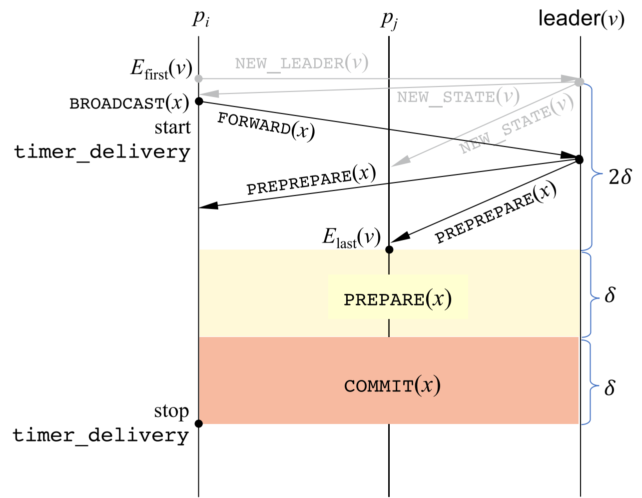

Fig. 7 An illustration of the bound on dur_delivery in Lemma 4

protocol the processes update their timeouts independently, and may thus disagree on their durations. For example, the first correct process p_i to detect a problem with the current view v will increase its timeouts and call `advance` (line 3). The synchronizer may then trigger `new_view` notifications at other correct processes before they detect the problem as well, so that their timeouts will stay unchanged (line 42).[2] One may think that this allows executions in which only some correct processes keep increasing their timeouts until they are high enough, whereas others are forever stuck with timeouts that are too low, invalidating the premise of Lemma 4. The following lemma rules out such scenarios and also trivially implies Lemma 4. It establishes that, in a sufficiently high view v with a correct leader, if the timeouts at a correct process p_i that enters v are high enough, then this process cannot be the first one to initiate a view change. Hence, for the protocol to enter another view, some other process with lower timeouts must call `advance` and thus increase their durations (line 3).

Lemma 5 *Let $v \geq \mathcal{V}$ be such that $E_{\text{first}}(v) \geq$ GST and* leader(v) *is correct, and consider a correct process p_i that enters v. If* dur_recovery$_i(v) > 6\delta$ *and* dur_delivery$_i(v) > 4\delta$ *then p_i is not the first correct process to call* `advance` *in v.*

Proof Since $E_{\text{first}}(v) \geq$ GST, messages sent by correct processes after $E_{\text{first}}(v)$ get delivered to all correct processes within δ and process clocks track real time. By contradiction, assume that p_i is the first correct process to call `advance` in v. This happens because a timer expires at p_i, and we now make a case split on which timer it is.

Assume first that timer_recovery expires at p_i. A process starts timer_recovery when it enters the view v (line 47), and hence, at $E_{\text{first}}(v)$ at the earliest (Fig. 6). Because p_i is the first correct process to call `advance` in v and dur_recovery$_i(v) > 6\delta$, no correct process calls `advance` in v until after $E_{\text{first}}(v) + 6\delta$. Then by Bounded Entry all correct processes enter v by $E_{\text{first}}(v) + 2\delta$. Also, by Validity no correct process can enter $v + 1$ until after $E_{\text{first}}(v) + 6\delta$, and by Proposition 1 the same holds for any view $> v$. Thus, all correct processes stay in v at least until $E_{\text{first}}(v) + 6\delta$. This implies that no correct process will send a message with a view $> v$ before $E_{\text{first}}(v) + 6\delta$. Therefore, in the messages exchanges we describe below correct processes will not discard any messages with view v before this time (see §4.5).

When a correct process enters v, it sends a `NEW_LEADER` message to the leader of v, which happens by $E_{\text{first}}(v) + 2\delta$. When the leader receives such messages from a quorum of processes, it broadcasts a `NEW_STATE` message. Thus, by $E_{\text{first}}(v) + 4\delta$ all correct processes receive this message and set status $=$ NORMAL. If at that point init_log_length $\leq$ last_delivered at p_i, then the process stops timer_recovery (line 66), which contradicts our assumption. Hence, init_log_length $>$ last_delivered. When a correct process receives `NEW_STATE`, it sends `PREPARE` messages for all positions $\leq$ init_log_length (line 62). It then takes the correct processes at most 2δ to exchange the sequence of `PREPARE` and `COMMIT` messages that commits the values at all positions $\leq$ init_log_length. Thus, by $E_{\text{first}}(v) + 6\delta$ the process p_i commits and delivers all these positions, stopping timer_recovery (line 41): a contradiction.

---

[2] Recall that even a single `advance` call at a correct process may lead to a view switch (§3). For example, with the synchronizer in Fig. 3 this may happen as follows. The `advance` call at a process generates `WISH`$(v + 1)$ (line 2) and the Byzantine processes produce another f copies of the same message. Correct processes receiving the resulting $f+1$ copies of the message relay it via line 8, which yields $2f+1$ copies in total. The synchronizer then triggers `new_view`$(v+1)$ notifications at correct processes that receive all these copies (line 15), causing them to enter $v + 1$ without increasing their timeouts.

It remains to consider the case when, for some value x, $\mathsf{timer_delivery}[x]$ expires at p_i in v. The process starts $\mathsf{timer_delivery}[x]$ when it receives $\mathtt{BROADCAST}(x)$ and it has not yet delivered x (line 14). Let t be the time when this happens; then $t \geq E_{\mathrm{first}}(v)$ (Fig. 7). Because p_i is the first correct process to call advance in v and $\mathsf{dur_delivery}_i(v) > 4\delta$, no correct process calls advance in v until after $t + 4\delta$. Then by Bounded Entry all correct processes enter v by $E_{\mathrm{first}}(v) + 2\delta$. Furthermore, by Validity no correct process can enter $v + 1$ until after $t + 4\delta$, and by Proposition 1 the same holds for any view $> v$. Thus, all correct processes stay in v at least until $t + 4\delta$.

The process p_i has $\mathsf{status} = \mathrm{NORMAL}$ at t, so that by this time p_i has handled the NEW_STATE message from the leader of v. Thus, all correct processes receive NEW_STATE by $t + \delta$. Since $t \geq E_{\mathrm{first}}(v)$ and all correct processes enter v by $E_{\mathrm{first}}(v) + 2\delta$, all correct processes handle NEW_STATE by $t + 2\delta$. When a process handles $\mathtt{NEW_STATE}(v, _, _)$, it sends PREPARE messages for all positions $\leq$ init_log_length. Therefore, by $t + 2\delta$ all correct processes send PREPARE for all positions $\leq$ init_log_length.

When p_i starts $\mathsf{timer_delivery}[x]$, it sends $\mathtt{FORWARD}(x)$ to $\mathsf{leader}(v)$, which receives the message no later than $t + \delta$. Consider first the case when $\mathsf{leader}(v)$ has x in its log at position $k \leq$ init_log_length when it receives $\mathtt{FORWARD}(x)$. Then all correct processes send PREPARE for all positions $\leq k$ by $t + 2\delta$. Assume now that, when the leader receives $\mathtt{FORWARD}(x)$, either $x \notin \mathsf{log}$ or for some $k >$ init_log_length we have $\mathsf{log}[k] = x$. In the former case the leader sends $\mathtt{PREPREPARE}(v, k, x)$ to all processes. In the latter case, due to lines 52 and 19, the leader has already sent $\mathtt{PREPREPARE}(v, k, x)$ to all processes. Thus, in either case the leader sends $\mathtt{PREPREPARE}(v, k, x)$ no later than $t + \delta$. Hence, due to lines 52 and 19, the leader sends a PREPREPARE for all positions from init_log_length $+ 1$ up to k no later than $t + \delta$, and all correct processes receive these messages no later than $t + 2\delta$. We have established that all correct processes have handled $\mathtt{NEW_STATE}(v, _, _)$ by $t + 2\delta$. Then all correct processes handle the PREPREPARE messages for positions from init_log_length $+ 1$ up to k by $t + 2\delta$, i.e., they send a PREPARE for each of these positions. Furthermore, we have established that all correct processes send a PREPARE message for each position $\leq$ init_log_length by $t + 2\delta$. Therefore, all correct processes send PREPARE for each position $\leq k$ by $t + 2\delta$. It then takes them at most 2δ to exchange the corresponding sequence of PREPARE and COMMIT messages. Hence, all correct processes receive COMMIT for all positions $\leq k$ by $t + 4\delta$.

Assume that when p_i receives all these COMMIT messages, it has last_delivered $< k$. Then p_i delivers x by $t + 4\delta$. Since p_i's $\mathsf{timer_delivery}[x]$ has not expired by then, the process stops the timer, which contradicts our assumption. Therefore, when p_i receives all the COMMIT messages for positions $\leq k$, it has last_delivered $\geq k$, so that p_i has already delivered a value x' at this position. Then p_i must have formed a certificate C' such that $\mathsf{committed}(C', v', k, \mathsf{hash}(x'))$. By Proposition 5, for some well-formed certificate C'' we have $\mathsf{prepared}(C'', v', k, \mathsf{hash}(x'))$. Since all correct processes stay in v until $t + 4\delta$, we must have $v' \leq v$. If $v = v'$, then by Proposition 3 we get $x = x'$. If $v' < v$, then by Lemma 2 we get $x = x'$. Hence, p_i delivers x. By line 12, p_i does not start $\mathsf{timer_delivery}[x]$ if x has already been delivered. Since p_i started the timer at t, it has to deliver x at some point after t and no later than $t + 4\delta$. Since p_i's $\mathsf{timer_delivery}[x]$ has not expired by then, the process stops the timer, which contradicts our assumption. □

Theorem 2 *PBFT-light satisfies the Liveness property of Byzantine atomic broadcast.*

Proof Consider a valid value x broadcast by a correct process. We first prove that x is eventually delivered by some correct process. By contradiction, assume that x is never delivered by a correct process. We now prove that in this case PBFT-light must keep switching views forever. We do this analogously to the proof of Proposition 2 in §3: we use Lemma 3 to show that correct processes keep calling advance, and the Progress property of the synchronizer to show that some of them will keep switching views as a result.

Claim 1 Every view is entered by some correct process.

Proof Since all correct processes call start (line 1), by Startup a correct process eventually enters some view. We now show that correct processes keep entering new views forever (analogously to the proof of Proposition 2 in §3). Assume that this is false, so that there exists a maximal view v entered by any correct process. Let P be any set of $f + 1$ correct processes and consider an arbitrary process $p_i \in P$ that enters v. The process that broadcast x is correct, and thus keeps broadcasting x until the value is delivered (line 11). Since x is never delivered, p_i is guaranteed to receive x while in v. Then by Lemma 3, p_i eventually calls advance while in v. Since p_i was picked arbitrarily, we have $\forall p_i \in P.\, E_i(v)\downarrow \implies A_i(v)\downarrow$. Then by Progress we get $E_{\mathrm{first}}(v + 1)\downarrow$, which yields a contradiction. Thus, correct processes keep entering views forever. The claim then follows from Proposition 1, ensuring that, if a view is entered by a correct process, then so are all preceding views. □

Let view v_1 be the first view such that $v_1 \geq \mathcal{V}$ and $E_{\mathrm{first}}(v_1) \geq \mathsf{GST}$; such a view exists by Claim 1. The next claim is needed to show that all correct processes will increase their timeouts high enough to satisfy the bounds in Lemma 4 (as per our previous discussion).

Claim 2 Every correct process calls the timer expiration handler (line 3) infinitely often.

Proof Assume the contrary and let C_{fin} and C_{inf} be the sets of correct processes that call the timer expiration handler finitely and infinitely often, respectively. Then $C_{\text{fin}} \neq \emptyset$, and by Claim 1 and Validity, $C_{\text{inf}} \neq \emptyset$. The values of $\mathsf{dur_delivery}$ and $\mathsf{dur_recovery}$ increase unboundedly at processes from C_{inf}, and do not change after some view v_2 at processes from C_{fin}. By Claim 1 and since leaders rotate round-robin, there is a view $v_3 \geq \max\{v_2, v_1\}$ with a correct leader such that any process $p_i \in C_{\text{inf}}$ that enters v_3 has $\mathsf{dur_delivery}_i(v_3) > 4\delta$ and $\mathsf{dur_recovery}_i(v_3) > 6\delta$. By Claim 1 and Validity, at least one correct process calls `advance` in v_3; let p_l be the first process to do so. Since $v_3 \geq v_2$, this process cannot be in C_{fin} because none of these processes increase their timers in v_3. Then $p_l \in C_{\text{inf}}$, contradicting Lemma 5. □

By Claims 1 and 2, there exists a view $v_4 \geq v_1$ with a correct leader such that some correct process enters v_4, and for any correct process p_i that enters v_4 we have $\mathsf{dur_delivery}_i(v_4) > 4\delta$ and $\mathsf{dur_recovery}_i(v_4) > 6\delta$. By Lemma 4, no correct process calls `advance` in v_4. Then, by Validity, no correct process enters $v_4 + 1$, which contradicts Claim 1. This contradiction shows that x must be delivered by a correct process. Then, since the protocol reliably broadcasts committed values (line 31), all correct processes will also eventually deliver x. From here the Liveness property follows. □

7 Latency analysis

Assume that PBFT-light is used with our SMR synchronizer in Fig. 3. We now quantify its latency, yielding the first detailed latency analysis for a PBFT-like protocol. To this end, we first show some latency bounds for our synchronizer. These are stated by the theorem below (proved in §C), which relies on the following notation. Given a view v that was entered by a correct process p_i, we let $T_i(v)$ denote the time at which p_i either attempts to advance from v or enters a view $> v$; we let $T_{\text{last}}(v)$ denote the latest time when a correct process does so. We assume that every correct process eventually attempts to advance from view 0 unless it enters a view > 0, i.e., $\forall p_i \in \mathcal{C}.\, T_i(0)\downarrow$.

Theorem 3 *Consider an execution with an eventual message delay δ. In this execution, the algorithm in Fig. 3 satisfies the following:*

A. $\forall v.\, E_{\text{first}}(v)\downarrow \;\wedge\; A_{\text{first}}(0) < \mathsf{GST} \implies E_{\text{last}}(v) \leq \max(E_{\text{first}}(v), \mathsf{GST} + \rho) + 2\delta$.
B. $\forall v.\, E_{\text{first}}(v+1)\downarrow \implies E_{\text{last}}(v+1) \leq \begin{cases} \max(T_{\text{last}}(v), \mathsf{GST} + \rho) + \delta, & \textit{if } A_{\text{first}}(0) < \mathsf{GST}; \\ T_{\text{last}}(v) + \delta, & \textit{otherwise.} \end{cases}$

Property A in the theorem bounds the latest time any correct process can enter a view that has been previously entered by a correct process. It is similar to Bounded Entry, but also handles views $< \mathcal{V}$. Property B refines Progress: while the latter guarantees that the synchronizer will enter $v + 1$ if enough processes ask for this, the former bounds the time by which this will happen.

To quantify the latency of PBFT-light using the above theorem, we assume the existence of a known upper bound Δ on the maximum value of δ in any execution [42, 52], so that we always have $\delta < \Delta$. In practice, Δ provides a conservative estimate of the message delay during synchronous periods, which may be much higher than the maximal delay δ in a particular execution. We modify the protocol in Figure 4 so that in lines 7-8 it does not increase $\mathsf{dur_recovery}$ and $\mathsf{dur_delivery}$ above 6Δ and 4Δ, respectively. This corresponds to the bounds in Lemma 4 and preserves the protocol liveness. Finally, we assume that periodic handlers (line 4 in Figure 3 and line 11 in Figure 4) are executed every ρ time units, and that the latency of reliable broadcast in line 31 under synchrony is $\leq \delta + \rho$ (this corresponds to an implementation that just periodically retransmits `DECISION` messages).

We quantify the latency of PBFT-light in both bad and good cases. For the bad case we assume that the protocol starts during the asynchronous period. Given a value x broadcast before GST, we quantify how quickly after GST all correct processes deliver x. For simplicity, we assume that timeouts are high enough at GST and that $\mathsf{leader}(\mathcal{V})$ is correct.

Theorem 4 *Assume that before* GST *all correct processes start executing the protocol and one of them broadcasts x. Let $\mathcal{V}$ be defined as in Theorem 1 and assume that* $\mathsf{leader}(\mathcal{V})$ *is correct and at* GST *each correct process has* $\mathsf{dur_recovery} > 6\delta$ *and* $\mathsf{dur_delivery} > 4\delta$. *Then all correct processes deliver x by*

$$\mathsf{GST} + \rho + \max\{\rho + \delta, 6\Delta\} + 4\Delta + \max\{\rho, \delta\} + 7\delta.$$

Although the latency bound looks complex, its main message is simple: PBFT-light recovers after a period of asynchrony in bounded time. This time is dominated by multiples of Δ; without the assumption that $\mathsf{leader}(\mathcal{V})$ is correct it would also be multiplied by f due to going over up to f views with faulty leaders.

We give the proof of Theorem 4 in §7.1 below. Here we informally highlight how its proof exploits the latency guarantees of our synchronizer (Theorem 1). The bound in Theorem 4 has to account for an unfavorable scenario where the last view $\mathcal{V} - 1$ of the asynchronous period is not operational (e.g., not all correct processes enter it). In this case, to deliver x the protocol first needs to bring all correct processes into the same view $\mathcal{V}$. To bound the time required for that,

we first use Property A to determine the latest time when a correct process can enter $\mathcal{V}-1$: $\mathsf{GST}+\rho+2\delta$. We then add the time such a process may spend in $\mathcal{V}-1$ before it detects that x is taking too long to get delivered and call `advance`: $\max\{\rho+\delta, 6\Delta\}+4\Delta$, where 6Δ and 4Δ come from the maximal timeout values. The first clause of Property B then shows that all correct processes will enter $\mathcal{V}$ within an additional δ, i.e.,

$$E_{\text{last}}(\mathcal{V}) \le \mathsf{GST}+\rho+\max\{\rho+\delta, 6\Delta\}+4\Delta+3\delta.$$

Finally, we add the time for x to be delivered in $\mathcal{V}$. It takes $\max\{\rho+\delta, 2\delta\}$ for the leader of $\mathcal{V}$ to get x and switch its status to NORMAL, and another 3δ to have it delivered, for the total of $\max\{\rho+\delta, 2\delta\}+3\delta = \max\{\rho, \delta\}+4\delta$.

We now consider the case when the protocol starts during the synchronous period, i.e., after GST. The following theorem quantifies how quickly all correct processes enter the first functional view, which in this case is view 1. If $\mathsf{leader}(1)$ is correct, it also quantifies how quickly a broadcast value x is delivered by all correct processes. The bound takes into account the following optimization: in view 1 the processes do not need to exchange `NEW_LEADER` messages. Then, after the systems starts up, the protocol delivers values within 4δ, which matches an existing lower bound of 3δ for the delivery time starting from the leader [3]. The proof of the theorem (given in §7.1) exploits the second clause of Property B.

Theorem 5 *Assume that all correct processes start the protocol after* GST *with* $\mathsf{dur_recovery} > 5\delta$ *and* $\mathsf{dur_delivery} > 4\delta$. *Then the* $\mathcal{V}$ *defined in Theorem 1 is equal to 1 and* $E_{\text{last}}(1) \le T_{\text{last}}(0)+\delta$. *Furthermore, if a correct process broadcasts* x *at* $t \ge \mathsf{GST}$ *and* $\mathsf{leader}(1)$ *is correct, then all correct processes deliver* x *by* $\max\{t, T_{\text{last}}(0)+\delta\}+4\delta$.

7.1 Proof of the latency bounds for PBFT-light

We start with a simple lemma, whose proof we omit. The lemma gives an upper bound on the time it takes for all correct processes to deliver a value once this has been delivered by a correct process.

Lemma 6 *If a correct process delivers a value* x *at* t, *then all correct processes deliver* x *by* $\max\{t+\delta, \mathsf{GST}+\rho+\delta\}$.

The next lemma generalizes Lemma 3. It shows that if a process broadcast a value but it does not deliver it for a sufficiently long period of time, then the process will call advance or move to a higher view. Furthermore, the lemma also bounds the time the process takes to do so.

Lemma 7 *Assume that a correct process* p_i *that enters* v *receives* `BROADCAST`(x) *at* $t \ge E_i(v)$ *for a valid value* x. *Assume that* p_i *does not deliver* x *until after* $\max\{t, E_i(v)+\mathsf{dur_recovery}_i(v)\}+\mathsf{dur_delivery}_i(v)$. *Then* $T_i(v)\downarrow$ *and*

$$T_i(v) \le \max\{t, E_i(v)+\mathsf{dur_recovery}_i(v)\} + \mathsf{dur_delivery}_i(v).$$

Proof We know that at some point p_i enters view v, and at this moment it starts timer_recovery. If the timer expires, then p_i calls `advance` in v by $E_i(v)+\mathsf{dur_recovery}_i(v)$, as required. Assume that timer_recovery does not expire at p_i. Then p_i stops the timer at lines 4, 41, 66 or 43. If p_i stops the timer at line 4, then it calls `advance` in v by $E_i(v)+\mathsf{dur_recovery}_i(v)$, as required. If p_i stops the timer at line 43, then it enters a higher view by $E_i(v)+\mathsf{dur_recovery}_i(v)$, as required. Assume now that p_i stops the timer at lines 41 or 66. This implies that p_i sets status = NORMAL by $E_i(v)+\mathsf{dur_recovery}_i(v)$ while in v. If p_i calls `advance` or enters a higher view by $\max\{t, E_i(v)+\mathsf{dur_recovery}_i(v)\}$, we get the required. Assume that this is not the case. Then p_i sets status = NORMAL and receives x by $\max\{t, E_i(v)+\mathsf{dur_recovery}_i(v)\}$. Since p_i has not delivered x by $\max\{t, E_i(v)+\mathsf{dur_recovery}_i(v)\}$, by this point p_i starts timer_delivery$[x]$. If the timer expires, then p_i calls `advance` in v by $\max\{t, E_i(v)+\mathsf{dur_recovery}_i(v)\}+\mathsf{dur_delivery}_i(v)$, as required. Otherwise p_i stops the timer at lines 4, 39 or 43. If p_i stops the timer at line 43, then it enters a higher view by $\max\{t, E_i(v)+\mathsf{dur_recovery}_i(v)\}+\mathsf{dur_delivery}_i(v)$, as required. If p_i stops the timer at line 39, then it delivers x by $\max\{t, E_i(v)+\mathsf{dur_recovery}_i(v)\}+\mathsf{dur_delivery}_i(v)$, which is impossible. In the remaining case p_i calls `advance` in v by $\max\{t, E_i(v)+\mathsf{dur_recovery}_i(v)\}+\mathsf{dur_delivery}_i(v)$, as required. □

The next lemma bounds the latency of entering $\mathcal{V}$ given that all correct processes starts executing the protocol before GST and $\mathcal{V}=1$ in Theorem 3.

Lemma 8 *Assume that all correct processes start executing PBFT-light before* GST. *If* $\mathcal{V}=1$ *in Theorem 3, then* $E_{\text{first}}(\mathcal{V})\downarrow$ *and* $E_{\text{last}}(\mathcal{V}) \le \mathsf{GST}+\rho+\delta$.

Proof By Theorem 3, if $\mathcal{V}=1$, then $\mathsf{GV}(\mathsf{GST})=0$. Since all correct processes start executing the protocol before GST, then all correct processes attempt to advance from view 0 and $T_{\text{last}}(0) < \mathsf{GST}$. By Startup, $E_{\text{first}}(\mathcal{V})\downarrow$. Applying the first clause of Property B, we get $E_{\text{last}}(\mathcal{V}) \le \max\{T_{\text{last}}(0), \mathsf{GST}+\rho+\delta\}$. Since $T_{\text{last}}(0) < \mathsf{GST}$, then $E_{\text{last}}(\mathcal{V}) \le \mathsf{GST}+\rho+\delta$, as required. □

Finally, assuming that a value x is broadcast before any correct process enters a view v, we derive a bound on the time it takes for all correct processes to deliver x. The bound is given as a function of the latest time when some correct process enters v.

Lemma 9 *Consider a view $v \geq \mathcal{V}$ such that $E_{\text{first}}(v) \geq \mathsf{GST}$ and $\mathsf{leader}(v)$ is correct. Assume that a correct process p_j broadcast a value x before $E_{\text{first}}(v)$. If $\mathsf{dur_recovery}_i(v) > 6\delta$ and $\mathsf{dur_delivery}_i(v) > 4\delta$ at each correct process p_i that enters v, then all correct processes deliver x by $E_{\text{last}}(v) + \max\{\rho, \delta\} + 4\delta$.*

Proof By Lemma 4, no correct process calls `advance` in v. Therefore, by Validity, no correct process enters $v + 1$, and by Proposition 1 the same holds for any view $> v$. By Bounded Entry, all correct processes enter v. Assume that p_j delivers x by $E_{\text{last}}(v) + \rho$. Then by Lemma 6 all correct processes deliver x by $\max\{E_{\text{last}}(v) + \rho + \delta, \mathsf{GST} + \rho + \delta\}$. Since $E_{\text{first}}(v) \geq \mathsf{GST}$, we get $E_{\text{last}}(v)+\rho+\delta \geq \mathsf{GST}+\rho+\delta$. Thus, all correct processes deliver x by $E_{\text{last}}(v) + \rho + \delta$, as required.

Consider now the case when p_j does not deliver x by $E_{\text{last}}(v) + \rho$. Then p_j retransmits x between GST and $\mathsf{GST} + \rho \leq E_{\text{last}}(v) + \rho$, so that $\mathsf{leader}(v)$ receives x by $E_{\text{last}}(v) + \rho + \delta$. When a process enters v, it sends `NEW_LEADER` to $\mathsf{leader}(v)$. The leader receives $2f + 1$ of these messages by $E_{\text{last}}(v) + \delta$ and sends a `NEW_STATE` message to all processes. A process handles `NEW_STATE` by $E_{\text{last}}(v) + 2\delta$ and sets its status to NORMAL. Since no process calls `advance` in v, every correct process has $\mathsf{status} = \text{NORMAL}$ after handling `NEW_STATE` onwards. We have established that $\mathsf{leader}(v)$ receives x by $E_{\text{last}}(v)+\rho+\delta$. Then $\mathsf{leader}(v)$ sets $\mathsf{status} = \text{NORMAL}$ and receives x by $t \leq E_{\text{last}}(v) + \max\{\rho, \delta\} + \delta$. Assume that $\mathsf{leader}(v)$ has delivered x by $E_{\text{last}}(v)+\max\{\rho, \delta\}+\delta$. Then by Lemma 6 all correct process deliver x by $\max\{E_{\text{last}}(v) + \max\{\rho, \delta\} + 2\delta, \mathsf{GST} + \rho + \delta\}$. Since $E_{\text{first}}(v) \geq \mathsf{GST}$, we get $E_{\text{last}}(v) + \max\{\rho, \delta\} + 2\delta \geq \mathsf{GST} + \rho + \delta$. Thus, in this case all correct processes deliver x by $E_{\text{last}}(v) + \max\{\rho, \delta\} + 2\delta$, as required.

Assume now that $\mathsf{leader}(v)$ has not delivered x by t. Consider first then case when $\mathsf{leader}(v)$ already has x in its log at t because x was prepared in a previous view. Then all correct processes send $\texttt{PREPARE}(v, k, \mathsf{hash}(x))$ for a position k by $E_{\text{last}}(v) + 2\delta$. Consider now the case when $\mathsf{leader}(v)$ either has x in its log at t because it was already proposed in v; or $\mathsf{leader}(v)$ does not have it. In this case, it follows that $\mathsf{leader}(v)$ sends $\texttt{PREPREPARE}(v, k, x)$ to all correct processes by t, which all correct processes receive by $t + \delta$. When a correct process receives $\texttt{PREPREPARE}(v, k, x)$, it sends $\texttt{PREPARE}(v, k, \mathsf{hash}(x))$. Thus, all correct process send $\texttt{PREPARE}(v, k, \mathsf{hash}(x))$ by $t+\delta$. Since $t \leq E_{\text{last}}(v)+\max\{\rho, \delta\}+\delta$. Then, in both cases, all correct processes send $\texttt{PREPARE}(v, k, \mathsf{hash}(x))$ by $E_{\text{last}}(v) + \max\{\rho, \delta\} + 2\delta$. It then takes the correct processes at most 2δ to exchange the sequence of `PREPARE` and `COMMIT` messages that commit x. Therefore, all correct processes commit x by $E_{\text{last}}(v) + \max\{\rho, \delta\} + 4\delta$. Let p_i be a correct process. Because $\mathsf{leader}(v)$ is correct, we can show that $\mathsf{last_delivered} \geq k-1$ at p_i by $E_{\text{last}}(v) + \max\{\rho, \delta\} + 4\delta$. If $\mathsf{last_delivered} = k - 1$, then p_i delivers x by $E_{\text{last}}(v) + \max\{\rho, \delta\} + 4\delta$. If $\mathsf{last_delivered} > k - 1$ at p_i by $E_{\text{last}}(v) + \max\{\rho, \delta\} + 4\delta$, then we can show that p_i has already delivered x before. Since p_i was picked arbitrarily, we can conclude that all correct processes deliver x by $E_{\text{last}}(v) + \max\{\rho, \delta\} + 4\delta$, as required. □

Proof of Theorem 4 Let p_j be the correct process that broadcast x, and let $\mathcal{V}$ be defined as in Theorem 1. Assume first that $\mathcal{V} > 1$. We have $A_{\text{first}}(0) < \mathsf{GST}$ and $E_{\text{first}}(\mathcal{V}-1) < \mathsf{GST}+\rho$. Then by Property A,

$$E_{\text{last}}(\mathcal{V} - 1) \leq \mathsf{GST} + \rho + 2\delta. \tag{5}$$

If at least one correct process p_i delivers x by $\mathsf{GST} + \rho + 2\delta + \max\{\rho + \delta, 6\Delta\} + 4\Delta$, then by Lemma 6, all correct processes deliver x by $\mathsf{GST}+\rho+3\delta+\max\{\rho+\delta, 6\Delta\}+4\Delta$, as required. Assume now that no correct process p_i delivers x by $\mathsf{GST}+\rho+2\delta+\max\{\rho+\delta, 6\Delta\}+4\Delta$. In particular, this implies that p_j has not delivered x by $\mathsf{GST}+2\rho+2\delta$, so that it retransmits x between $\mathsf{GST} + \rho + 2\delta$ and $\mathsf{GST} + 2\rho + 2\delta$. Consider a correct process p_i that enters $\mathcal{V} - 1$ and let t_i be the time when this process receives the $\texttt{BROADCAST}(x)$ retransmission from p_j; then

$$t_i \leq \mathsf{GST} + 2\rho + 3\delta \tag{6}$$

and by (5),

$$t_i > \mathsf{GST} + \rho + 2\delta \geq E_i(\mathcal{V} - 1). \tag{7}$$

We now obtain:

$$\begin{aligned}
&\max\{t_i, E_i(\mathcal{V} - 1) + \mathsf{dur_recovery}_i(\mathcal{V} - 1)\} + \\
&\quad \mathsf{dur_delivery}_i(\mathcal{V} - 1) \\
&\quad \leq \max\{t_i, E_i(\mathcal{V} - 1) + 6\Delta\} + 4\Delta \\
&\qquad\qquad\qquad \text{since } \mathsf{dur_recovery}_i(\mathcal{V} - 1) \leq 6\Delta \\
&\qquad\qquad\qquad \text{and } \mathsf{dur_delivery}_i(\mathcal{V} - 1) \leq 4\Delta \\
&\quad \leq \max\{\mathsf{GST} + 2\rho + 3\delta, E_i(\mathcal{V} - 1) + 6\Delta\} + 4\Delta \\
&\qquad\qquad\qquad \text{by (6)} \\
&\quad \leq \max\{\mathsf{GST} + 2\rho + 3\delta, \mathsf{GST} + \rho + 2\delta + 6\Delta\} + 4\Delta \\
&\qquad\qquad\qquad \text{by (5)} \\
&\quad = \mathsf{GST} + \rho + 2\delta + \max\{\rho + \delta, 6\Delta\} + 4\Delta
\end{aligned}$$

Then, since we assume that no correct process delivers x by $\mathsf{GST}+\rho+2\delta+\max\{\rho+\delta, 6\Delta\}+4\Delta$, by (7) and Lemma 7 we get $T_i(\mathcal{V} - 1)\downarrow$. Thus, p_i either calls `advance` in $\mathcal{V} - 1$ or enters a higher view. If at least one correct process enters a view higher than $\mathcal{V}$, then by Proposition 1, $E_{\text{first}}(\mathcal{V})\downarrow$. If all correct processes that enter $\mathcal{V} - 1$ call `advance`, then

by Progress we get $E_{\text{first}}(\mathcal{V})\downarrow$ as well. By Lemma 7, we also have

$$T_i(\mathcal{V}-1) \le \max\{t_i, E_i(\mathcal{V}-1) + \mathsf{dur_recovery}_i(\mathcal{V}-1)\} + \mathsf{dur_delivery}_i(\mathcal{V}-1) \le \mathsf{GST} + \rho + 2\delta + \max\{\rho+\delta, 6\Delta\} + 4\Delta$$

for any correct process p_i that enters $\mathcal{V}-1$. Applying the first clause of Property B, we get

$$E_{\text{last}}(\mathcal{V}) \le \max\{T_{\text{last}}(\mathcal{V}-1), \mathsf{GST}+\rho\} + \delta \le \mathsf{GST} + \rho + 3\delta + \max\{\rho+\delta, 6\Delta\} + 4\Delta. \quad (8)$$

Thus, if $\mathcal{V} > 1$, then either all correct processes deliver x by $\mathsf{GST}+\rho+2\delta+\max\{\rho+\delta, 6\Delta\}+4\Delta$, or $E_{\text{first}}(\mathcal{V})\downarrow$ and (8) holds. Furthermore, if $\mathcal{V} = 1$, then by Lemma 8, $E_{\text{first}}(\mathcal{V})\downarrow$ and $E_{\text{last}}(\mathcal{V}) \le \mathsf{GST} + \rho + \delta$. We have thus established that either all correct processes deliver x by $\mathsf{GST} + \rho + 2\delta + \max\{\rho+\delta, 6\Delta\} + 4\Delta$, or $E_{\text{first}}(\mathcal{V})\downarrow$ and (8) holds. In the latter case, by Lemma 9, all correct processes deliver x by $E_{\text{last}}(\mathcal{V}) + \max\{\rho, \delta\} + 4\delta$, and by (8),

$$E_{\text{last}}(\mathcal{V}) + \max\{\rho, \delta\} + 4\delta \le \mathsf{GST} + \rho + \max\{\rho+\delta, 6\Delta\} + 4\Delta + \max\{\rho, \delta\} + 7\delta,$$

as required. □

The following lemma shows, considering the special case of $v = \mathcal{V} = 1$, that when the protocol starts during the synchronous period and timers a long enough at each correct process, then no process calls `advance` in view 1.

Lemma 10 *Assume that $\mathcal{V} = 1$ in Theorem 3, $E_{\text{first}}(1) \ge \mathsf{GST}$ and $\mathsf{leader}(1)$ is correct. If $\mathsf{dur_recovery}_i(1) > 5\delta$ and $\mathsf{dur_delivery}_i(1) > 4\delta$ at each correct process p_i that enters view 1, then no correct process calls* `advance` *in view 1.*

We omit the proof of this lemma. It is virtually identical to that of Lemma 4, considering the special case of $v = \mathcal{V} = 1$ and the optimization by which in view 1 the processes do not exchange `NEW_LEADER` messages.

The next lemma gives the core argument for the proof of Theorem 5. It uses Lemma 10 to show that, under the same conditions, if a correct process broadcast a value, then all correct processes will deliver it. The lemma also bounds the time it take for all correct processes to do so.

Lemma 11 *Assume that $\mathcal{V} = 1$ in Theorem 3, $E_{\text{first}}(1) \ge \mathsf{GST}$ and $\mathsf{leader}(1)$ is correct. Assume that a correct process p_j broadcast a value x at $t \ge \mathsf{GST}$. If $\mathsf{dur_recovery}_i(1) > 5\delta$ and $\mathsf{dur_delivery}_i(1) > 4\delta$ at each correct process p_i that enters 1, then all correct processes deliver x by $\max\{t, E_{\text{last}}(1)\} + 4\delta$.*

Proof By Lemma 10, no correct process calls `advance` in 1. Therefore, by Validity, no correct process enters 2, and by Proposition 1 the same holds for any view > 1. By Bounded Entry, all correct processes enter 1.

When $\mathsf{leader}(1)$ enters view 1, it sends a `NEW_STATE` message to all processes. A process handles `NEW_STATE` by $E_{\text{last}}(1) + \delta$ and sets its status to NORMAL. The $\mathsf{leader}(1)$ receives x by $t + \delta$. The leader will send `PREPREPARE`$(1, k, x)$ when it has received x and has $\mathsf{status} =$ NORMAL, i.e., by $\max\{t, E_{\text{last}}(1)\} + \delta$. All correct processes receive `PREPREPARE`$(1, k, x)$ by $\max\{t, E_{\text{last}}(1)\} + 2\delta$. It then takes the correct processes at most 2δ to exchange the sequence of `PREPARE` and `COMMIT` messages that commit x. Therefore, all correct processes commit x by $\max\{t, E_{\text{last}}(1)\} + 4\delta$. Let p_i be a correct process. Because $\mathsf{leader}(1)$ is correct, we can show that $\mathsf{last_delivered} \ge k-1$ at p_i by $\max\{t, E_{\text{last}}(1)\} + 4\delta$. If $\mathsf{last_delivered} = k-1$, then p_i delivers x by $\max\{t, E_{\text{last}}(1)\} + 4\delta$. In case $\mathsf{last_delivered} > k-1$ at p_i by $\max\{t, E_{\text{last}}(1)\} + 4\delta$, we can show that p_i has already deliver x before. Since p_i was picked arbitrarily, we can conclude that all correct processes deliver x by $\max\{t, E_{\text{last}}(1)\} + 4\delta$, as required. □

Proof of Theorem 5 When a correct process starts the protocol, it calls `advance` from view 0 unless it has already entered a higher view. If all correct processes call `advance` from view 0 when they start the protocol, then by Startup we get $E_{\text{first}}(1)\downarrow$. If at least one correct process does not call `advance` from view 0 because it has already entered a higher view, then by Proposition 1 we also get $E_{\text{first}}(1)\downarrow$. Since all correct processes start the protocol after GST, we have $A_{\text{first}}(0) > \mathsf{GST}$. Thus, applying the second clause of Property B, we get $E_{\text{last}}(1) \le T_{\text{last}}(0) + \delta$. Furthermore, by Theorem 3 we get $\mathcal{V} = 1$. Hence, $E_{\text{last}}(\mathcal{V}) \le T_{\text{last}}(0) + \delta$, as required. By Lemma 11, all correct processes deliver x by $\max\{t, E_{\text{last}}(\mathcal{V})\} + 4\delta \le \max\{t, T_{\text{last}}(0) + \delta\} + 4\delta$, as required. □

8 Additional case studies

To demonstrate the generality of the SMR synchronizer abstraction, we have also used it to ensure the liveness of two other protocols. First, we handle a variant of PBFT that periodically forces a leader change, as is common in modern Byzantine SMR [28, 57, 58]. In this protocol a process calls `advance` not only when it suspects the current leader to be faulty, but also when it delivers B values proposed by this leader (for a fixed B). For this variant of PBFT we have also established latency bounds when using the synchronizer in Fig. 3. Second, we have applied the SMR synchronizer abstraction to a variant of the above protocol that follows the approach of HotStuff [59]. The resulting protocol adds

an extra communication step to the normal path of PBFT in exchange for reducing the communication complexity of leader change. We defer the details about these two protocols to [15, Appendices D-E]. Their liveness proofs follow the methodology we proposed for PBFT-light, establishing analogs of Lemmas 3–5.

9 Related work and discussion

9.1 Failure detectors

Failure detectors and leader oracles [25, 35] have been widely used for implementing consensus and SMR under benign failures [39, 40, 49], but their implementations under Byzantine failures are either impractical [45] or detect only restricted failure types [31, 41, 48].

9.2 Other liveness abstractions

As an alternative to failure detection, the abstractions proposed in §5.6.2 of the textbook [19] and [11, 56] rely on client hints (similar to our `advance` primitive) to indicate potential leader failures.

The *Byzantine Epoch-Change (BEC)* abstraction of [19, §5.6.4] divides computation into a series of epochs (similar to our views) each of which is associated with a unique leader. The processes running a consensus algorithm on top of BEC may "complain" about the lack of progress by the current leader via a special call. Unfortunately, a key property of BEC, *Eventual Leadership*, is too strong: similarly to leader oracles, it requires all correct processes to eventually enter the same epoch with a correct leader. If the number of correct processes in some non-empty set C complaining about a faulty leader of some epoch is $\leq f$, and the remaining $\geq f+1$ correct processes in a set C' observe the leader to behave correctly, the total number of complaints will not be sufficient to force the correct processes to leave this epoch. This results in a liveness bug in the BEC-based Byzantine consensus protocol of [19, §5.6.4]: the processes in C may never be able to reach a decision if the faulty leader chooses to only communicate with the processes in C' in every phase of the consensus protocol.[3] Although this bug can be easily fixed by reliably broadcasting decisions (as we do in PBFT-light, see §4), the Eventual Leadership property will remain broken, since correct processes will be allowed to remain forever in an epoch with a faulty leader.

Although our `advance` is similar to "complain", we use it to implement a weaker abstraction of an SMR synchronizer. We then obtain properties similar to accuracy and completeness of failure detectors by carefully combining SMR-level timers with uses of `advance` (Lemmas 3–4). Also, while [19] does not specify constraints on the use of "complain" (see [15, §A] for details), we give a complete characterization of `advance` and show its sufficiency for solving SMR.

BFT-SMaRt [11, 56] built on the ideas of [19] to propose an abstraction of *validated and provable consensus (VP-Consensus)*, which allows its clients to control leader changes via a special *VP-Timeout* call. Although the overall BFT-SMaRt protocol appears to be correct, the proposed VP-Consensus abstraction is underspecified. In particular, its termination relies on the assumption that the clients leave enough time in-between consecutive *VP-Timeout* calls to allow a correct leader to complete the consensus protocol. This in turn requires knowledge about the time necessary for such a decision, which is implementation-dependent.

A more detailed discussion of the above abstractions and their properties can be found in [15, §A].

9.3 Emulating synchrony

Alternative abstractions avoid dependency on the specifics of a failure model by simulating synchrony [12, 26, 36, 44]. The first such abstraction is due to Awerbuch [8] who proposed a family of synchronizer algorithms emulating a round-based synchronous system of top of an asynchronous network with reliable communication and processes. The first such emulation in a failure-prone partially synchronous system was introduced in the DLS paper [33]. It relied on an expensive clock synchronization protocol, which interleaved its messages with every step of a high-level consensus algorithm implemented on top of it. Later work proposed more practical solutions, which reduce the synchronization frequency by relying on either timers [32] or synchronized hardware clocks [1, 5, 37] (the latter can be obtained using one of the existing fault-tolerant clock synchronization algorithms [30, 55]). However, the DLS model emulates communication-closed rounds, i.e., eventually, a process in a round r receives *all* messages sent by correct processes in r. This property rules out *optimistically responsive* [53, 59] protocols such as PBFT, which can make progress as soon as they receive messages from *any quorum*.

9.4 Consensus synchronizers

To address the shortcoming of DLS rounds, recent work proposed a more flexible abstraction ("consensus synchronizer" in §A) that switches processes through an infinite series of *views* [14, 16, 27, 51, 59]. In contrast to rounds, each view may subsume multiple communication steps, and its duration can either be fixed in advance [51] or grow dynamically [16, 59]. Although consensus synchronizers can be used for efficient single-shot Byzantine consensus [16], using them for SMR results in suboptimal implementations. A classical

[3] The bug was acknowledged by the textbook authors [18].

approach is to decide on each SMR command using a separate black-box consensus instance [54]. However, implementing the latter using a consensus synchronizer would force the processes in every instance to iterate over the same sequence of potentially bad views until the one with a correct leader and sufficiently long duration could be reached.

An alternative approach was proposed in HotStuff [59]. This SMR protocol is driven by a *pacemaker*, which keeps generating views similarly to a consensus synchronizer. Within each view HotStuff runs a voting protocol that commits a block of client commands in a growing hash chain. Although the voting protocol is optimistically responsive, committing the next block is delayed until the pacemaker generates a new view, which increases latency. The cost the pacemaker may incur to generate a view is also paid for every single block.

Recently, Civit et al. [27] proposed RareSync, a consensus synchronizer designed as a building block for a communication efficient partially synchronous Byzantine consensus protocol. To reduce communication complexity, in RareSync processes only exchange messages to synchronize every $(f+1)$st view, and rely entirely on their local clocks to synchronize the remaining views. Since implementing this approach requires the view synchronization schedule to be controlled by the synchronizer, it cannot be used for implementing an SMR synchronizer where processes can explicitly request a view change via `advance`. Also, in contrast to our synchronizer, the complexity guarantees of RareSync critically depend on the assumption that every message sent at time t is delivered within $\max\{\mathsf{GST}, t\} + \delta$. This assumption is not a part of the original partial synchrony models [33] and is unrealistic to require in practice.

9.5 SMR synchronizers

In contrast to the above approaches, SMR synchronizers allow the application to initiate view changes on demand via an `advance` call. As we show, this affords SMR protocols the flexibility to judiciously manage their view synchronization schedule: in particular, it prevents the timeouts from growing unnecessarily (§8) and avoids the overheads of further view synchronizations once a stable view is reached (Lemma 4, §6).

The first synchronizer with a `new_view`/`advance` interface, which here we call an SMR synchronizer, was proposed by Naor et al. [50, 51]. They used it as an intermediate module in a communication-efficient implementation of a consensus synchronizer. The latter is sufficient to ensure the liveness of HotStuff [59] via either of the two straightforward SMR constructions we described above. The specification of the `new_view`/`advance` module of Naor et al. was only used as a stepping stone in the proof of their consensus synchronizer, and as a result, is more low-level and complex than our SMR synchronizer specification. Naor et al. did not investigate the usability of the SMR synchronizer abstraction as a generic building block applicable to a range of Byzantine SMR protocols—a gap we fill in this paper. Finally, they only handled a simplified version of partial synchrony where messages are never lost and δ is known a priori, whereas our SMR synchronizer implementation handles partial synchrony in its full generality. This implementation builds on a consensus synchronizer we previously proposed [16]. However, its correctness proof and performance analysis are more intricate, since the timing of the view switches is not fixed a priori, but driven by external `advance` inputs.

Aştefănoaei et al. [4] proposed another framework for implementing Byzantine SMR protocols, based on DLS rounds. This uses a simple synchronizer that does not exchange any messages: it recovers from a period of asynchrony by progressively increasing round durations until they are long enough for all correct processes to overlap in the same round. This way of view synchronization rules out optimistically responsive SMR protocols and does not bound the time to reach a decision after GST, as we do.

9.6 SMR liveness proofs

PBFT [22–24] is a seminal protocol whose design choices have been widely adopted [38, 46, 57, 58]. To the best of our knowledge, our proof in §6 is the first one to formally establish its liveness. An informal argument given in [24, §4.5.1] mainly justifies liveness assuming all correct processes enter a view with a correct leader and stay in that view for sufficiently long. It does not rigorously justify why such a view will be eventually reached, and in particular, how this is ensured by the interplay between SMR-level timeout management and view synchronization (§6). Liveness mechanisms were also omitted from the formal specification of PBFT by an I/O-automaton [22, 24]. PBFT and its follow-ups, such as Zyzzyva [46] and SBFT [38], often included mechanisms for view synchronization with a communication structure similar to Bracha broadcast [13], which is also used by our synchronizer. In contrast to these works, we separate the synchronizer into a reusable abstraction and rigorously prove its correctness, latency bounds and space requirements.

We have previously applied consensus synchronizers to several consensus protocols, including a single-shot version of PBFT [16]. These protocols and their proofs and are much more straightforward than the full SMR protocols we consider here. In particular, since a consensus synchronizer keeps switching processes between views regardless of whether their leaders are correct, the proof of the single-shot PBFT in [16] does not need to establish analogs of completeness and accuracy (Lemmas 3 and 4) or deal with the fact that processes may disagree on timeout durations (Lemma 5).

Byzantine SMR protocols often integrate view synchronization into the core protocol, enabling white-box optimizations [7, 17, 23, 29]. Our work does not rule out this approach, but allows making it more systematic: we can first develop efficient mechanisms for view synchronization independently from SMR protocols, and do white-box optimizations afterwards, if needed.

Overall, our synchronizer specification and the analysis of PBFT provide a blueprint for designing provably live and efficient Byzantine SMR protocols.

Funding This research was partially supported by the European Research Council (Starting Grant RACCOON), Nomadic Labs/Tezos Foundation, the Spanish Ministry of Science and Innovation (projects PRODIGY, DECO and BYZANTIUM), and the CHIST-ERA network (project REDONDA).

Data Availability Data sharing not applicable to this article as no datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors have no conflicts of interest to declare that are relevant to the content of this article.

A Constructing a consensus synchronizer from an SMR synchronizer

We now show that we can use an SMR synchronizer presented in this paper to implement a consensus synchronizer [16, 51] without extra overhead, thereby demonstrating the generality of the former abstraction. We first recap the interface and specification of a consensus synchronizer. This synchronizer produces notifications $\texttt{new_consensus_view}(v)$ at at each correct process, telling it to enter view v. A process can ensure that the consensus synchronizer has started operating by calling a special $\texttt{start}()$ function. We assume that each correct process eventually calls $\texttt{start}()$, unless it gets a $\texttt{new_consensus_view}$ notification first.

For a consensus protocol to terminate, its processes need to stay in the same view for long enough to complete the message exchange leading to a decision. Since the message delay δ after GST is unknown to the consensus protocol, we need to increase the view duration until it is long enough for the protocol to terminate. To this end, the synchronizer is parameterized by a function defining this duration – $F : \mathsf{View} \cup \{0\} \to \mathsf{Time}$, which is monotone, satisfies $F(0) = 0$, and increases unboundedly. The latter is formalized as follows:

$$\forall \theta. \exists v. \forall v'. v' \geq v \implies F(v') > \theta. \tag{9}$$

Figure 8 presents the specification of a consensus synchronizer proposed in [16]. This relies on the following

I. $\forall i, v, v'. \mathbb{E}_i(v)\downarrow \wedge \mathbb{E}_i(v')\downarrow \implies (v < v' \iff \mathbb{E}_i(v) < \mathbb{E}_i(v'))$
II. $\mathbb{E}_{\text{first}}(\mathcal{V}) \geq \mathsf{GST}$
III. $\forall i. \forall v \geq \mathcal{V}. p_i \in \mathcal{C} \implies \mathbb{E}_i(v)\downarrow$
IV. $\exists d. \forall v \geq \mathcal{V}. \mathbb{E}_{\text{last}}(v) \leq \mathbb{E}_{\text{first}}(v) + d$
V. $\forall v \geq \mathcal{V}. \mathbb{E}_{\text{first}}(v+1) > \mathbb{E}_{\text{first}}(v) + F(v)$

Fig. 8 Consensus synchronizer specification [16], holding for some $\mathcal{V} \in \mathsf{View}$

```
1 function start()
2   advance();

3 upon new_view(v)
4   stop_timer(timer_view);
5   start_timer(timer_view, F(v));
6   trigger new_consensus_view(v);

7 when timer_view expires
8   advance();
```

Fig. 9 A consensus synchronizer from an SMR synchronizer

notation, analogous to the one used for SMR synchronizers. Given a view v for which a correct process p_i received a $\texttt{new_consensus_view}(v)$ notification, we denote by $\mathbb{E}_i(v)$ the time when this happens; we let $\mathbb{E}_{\text{first}}(v)$ and $\mathbb{E}_{\text{last}}(v)$ denote respectively the earliest and the latest time when some correct process receives a $\texttt{new_consensus_view}(v)$ notification. Like an SMR synchronizer, a consensus synchronizer must guarantee that views only increase at a given process (Property I). A consensus synchronizer ensures view synchronization only starting from some view $\mathcal{V}$, entered after GST (Property II). Starting from $\mathcal{V}$, correct processes do not skip any views (Property III), enter each view $v \geq \mathcal{V}$ within at most d of each other (Property IV) and stay there until at least $F(v)$ after the first process enters v (Property V).

Figure 9 shows how we can construct a consensus synchronizer from an SMR synchronizer by generalizing the simple client in Fig. 2. Upon a $\texttt{start}()$ call, the consensus synchronizer just tells the underlying SMR synchronizer to advance (line 1). When the SMR synchronizer produces a $\texttt{new_view}(v)$ notification (line 3), the consensus synchronizer immediately produces the corresponding $\texttt{new_consensus_view}(v)$ notification (thus, we always have $\mathbb{E}_i(v) = E_i(v)$ and use the two interchangeably in the proofs below). The synchronizer also sets a timer $\mathsf{timer_view}$ for the duration $F(v)$. When the timer expires (line 7), the consensus synchronizer tells the SMR synchronizer to advance. We now prove that this construction is correct.

Theorem 6 *The consensus synchronizer in Fig. 9 satisfies the properties in Fig. 8, provided the SMR synchronizer it uses satisfies the properties in Fig. 1.*

First, analogously to Proposition 2 we can prove that the consensus synchronizer keeps switching processes through views forever.

Proposition 7 $\forall v.\, \exists v'.\, v' > v \wedge \mathbb{E}_{\text{first}}(v')\downarrow$.

The following lemma is used to prove Property V in Fig. 8.

Lemma 12 *If a correct process enters a view* $v > 0$ *and* $E_{\text{first}}(v) \geq \mathsf{GST}$*, then for all* $v' > v$*, no correct process attempts to advance from* $v' - 1$ *before* $E_{\text{first}}(v) + F(v)$.

Proof Suppose by contradiction that there exists a time $t' < E_{\text{first}}(v) + F(v)$ and a correct process p_i such that p_i attempts to advance from $v' - 1 > v - 1$ at t'. Since $v' \geq v + 1 > 1$, at t' the process p_i executes the handler at line 7 and the last view it entered is $v' - 1$. Since p_i.timer_view is not enabled at t', p_i must have entered $v' - 1$ at least $F(v)$ before t' according to its local clock. Since $v' - 1 \geq v$, by Proposition 1, we have $E_{\text{first}}(v' - 1) \geq E_{\text{first}}(v) \geq \mathsf{GST}$. Therefore, given that the clocks of all correct processes progress at the same rate as real time after GST, we get

$$E_{\text{first}}(v) \leq E_{\text{first}}(v' - 1) \leq t' - F(v' - 1).$$

Hence,

$$t' \geq E_{\text{first}}(v) + F(v' - 1).$$

Since F is non-decreasing and $v' - 1 \geq v$, we have $F(v' - 1) \geq F(v)$, so that

$$t' \geq E_{\text{first}}(v) + F(v),$$

which contradicts our assumption that $t' < E_{\text{first}}(v) + F(v)$. This contradiction shows the required. □

Proof of Theorem 6 Property I follows from Monotonicity of the SMR synchronizer. Let d and $\mathcal{V}$ be the witnesses for the existential from Bounded Entry and let $\mathcal{V}'$ be the minimal view such that $\mathcal{V}' \geq \mathcal{V}$, $E_{\text{first}}(\mathcal{V}') \geq \mathsf{GST}$ and $F(\mathcal{V}') \geq d$. Such a view exists by (9) and Proposition 7. Then Property II holds for $\mathcal{V} = \mathcal{V}'$. By Propositions 1 and 7, a correct process enters every view $v \geq \mathcal{V}'$. By Proposition 1, $v \geq \mathcal{V}'$ implies

$$E_{\text{first}}(v) \geq E_{\text{first}}(\mathcal{V}') \geq \mathsf{GST}. \tag{10}$$

Since F is a non-decreasing function, $F(v) \geq d$. Thus, by Lemma 12 and Bounded Entry, all correct processes enter v, and $E_{\text{last}}(v) \leq E_{\text{first}}(v) + d$, which validates Properties III and IV for $\mathcal{V} = \mathcal{V}'$. To prove Property V, fix a view $v \geq \mathcal{V}'$. Since a correct process enters view $v + 1$, by Validity, there exist a time $t < E_{\text{first}}(v + 1)$ at which some correct process attempts to advance from v. By (10), $E_{\text{first}}(v) \geq \mathsf{GST}$. Then by Lemma 12 we get $t \geq E_{\text{first}}(v) + F(v)$, so that $E_{\text{first}}(v + 1) > t \geq E_{\text{first}}(v) + F(v)$, as required. □

B Proof of the synchronizer correctness (Theorem 1)

The *local view* of a process p_i at time t, denoted $\mathsf{LV}_i(t)$, is the latest view entered by p_i at or before t, or 0 if p_i has not entered any views by then. We prove Validity using the following lemma, which shows that a $\texttt{WISH}(v + 1)$ can only be sent by a correct process if some correct process has already attempted to advance from the view v.

Lemma 13 *For all* t *and* $v \geq 0$*, if a correct process sends* $\texttt{WISH}(v + 1)$ *at* t*, then there exists a time* $t' \leq t$ *such that some correct process attempts to advance from* v *at* t'.

Proof We first prove the following auxiliary proposition:

$$\begin{aligned} &\forall p_i.\, \forall v.\, p_i \text{ is correct} \wedge p_i \text{ sends } \texttt{WISH}(v + 1) \text{ at } t \implies \\ &\quad \exists t' \leq t.\, \exists v' \geq v.\, \exists p_j. \\ &\quad p_j \text{ is correct} \wedge p_j \text{ attempts to advance from } v' \text{ at } t'. \end{aligned} \tag{11}$$

By contradiction, assume that a correct process p_i sends $\texttt{WISH}(v + 1)$ at t, but for all $t' \leq t$ and all $v' \geq v$, no correct process attempts to advance from v' at t'. Consider the earliest time t_k when some correct process p_k sends a $\texttt{WISH}(v_k)$ with $v_k \geq v + 1$, so that $t_k \leq t$.

Since p_k sends $\texttt{WISH}(v_k)$ at t_k, either $v_k = p_k.\mathsf{view}^+(t_k)$ or $p_k.\mathsf{view}(t_k) = p_k.\mathsf{view}^+(t_k) = v_k - 1$, and in the latter case p_k executes either line 2 or line 6. If $p_k.\mathsf{view}^+(t_k) = v_k \geq v + 1$, then $p_k.\mathsf{max_views}(t_k)$ includes $f + 1$ entries $\geq v_k \geq v + 1$, and therefore, there exists a correct process p_l that sent $\texttt{WISH}(v')$ with $v' \geq v + 1$ at $t_l < t_k$, contradicting the assumption that t_k is the earliest time when this can happen. Suppose that $p_k.\mathsf{view}(t_k) = p_k.\mathsf{view}^+(t_k) = v_k - 1$ and at t_k, p_k executes either line 2 or line 6. Then $\mathsf{LV}_k(t_k) = v_k - 1$. If p_k executes line 2 at t_k, then since $\mathsf{LV}_k(t_k) = v_k - 1$, p_k attempts to advance from $v_k - 1 \geq v$ at $t_k \leq t$, contradicting our assumption that no such attempt can occur. Suppose now that p_k executes the code in line 6 at t_k. If $v_k > 1$, then since $p_k.\mathsf{view}(t_k) = p_k.\mathsf{view}^+(t_k) = v_k - 1$, we know that $E_k(v_k - 1)$ is defined and satisfies $E_k(v_k - 1) < t_k$. Let $t'_k = E_k(v_k - 1)$ if $v_k > 1$, and $t'_k = 0$ otherwise. Then $p_k.\mathsf{view}(t'_k) = p_k.\mathsf{view}^+(t'_k) = v_k - 1$ and $p_k.\mathsf{advanced}(t'_k) = \textsc{false}$. Since $p_k.\mathsf{advanced}(t_k) = \textsc{true}$, there exists a time t''_k such that $t'_k < t''_k \leq t_k$ and p_k calls $\texttt{advance()}$ at t''_k. Since both p_k.view and $p_k.\mathsf{view}^+$ are non-decreasing, and both are equal to $v_k - 1$ at t''_k as well as t_k, $p_k.\mathsf{view}(t''_k) = p_k.\mathsf{view}^+(t''_k) = v_k - 1$. Thus, $\mathsf{LV}_k(t''_k) = v_k - 1$, which implies that at $t''_k < t_k \leq t$, p_k attempts to advance from $v_k - 1 \geq v$, contradicting our assumption that no such attempt can happen. Thus, (11) holds.

We now prove the lemma. Let t and v be such that some correct process sends $\texttt{WISH}(v + 1)$ at t. By (11), there exists a correct process that attempts to advance from a view $\geq v$

at or before t. Let t' be the earliest time when some correct process attempts to advance from a view $\geq v$, and let p_j be this process and $v' \geq v$ be the view from which p_j attempts to advance at t'. Thus, at t', p_j executes the code in line 2 and $\mathsf{LV}_j(t') = v' \geq v$. Hence, there exists an earlier time at which $p_j.\mathsf{view}^+ = p_j.\mathsf{view} = v'$. Since $p_j.\mathsf{view}^+$ is non-decreasing, $p_j.\mathsf{view}^+(t') \geq v'$. If $p_j.\mathsf{view}^+(t') > v'$, then given that $v' \geq v$, $p_j.\mathsf{view}^+(t') \geq v + 1$. Thus, there exists a correct process p_k and time $t'' < t'$ such that p_k sent $\mathtt{WISH}(v'')$ with $v'' \geq v + 1$ to p_j at t''. By (11), there exists a time $\leq t'' < t'$ at which some correct process attempts to advance from a view $\geq v'' - 1 \geq v$, which is impossible. Thus, $p_j.\mathsf{view}^+(t') = v'$. Since $\mathsf{LV}_j(t') = v'$, we have $p_j.\mathsf{view}(t') = p_j.\mathsf{view}^+(t') = v' \geq v$. By the definitions of view and view$^+$, v' is both the lowest view among the highest $2f + 1$ views in $p_j.\mathsf{max_views}(t')$, and the lowest view among the highest $f + 1$ views in $p_j.\mathsf{max_views}(t')$. Hence, $p_j.\mathsf{max_views}(t')$ includes $f + 1$ entries equal to v', and therefore, there exists a correct process p_k such that

$$p_j.\mathsf{view}(t') = p_j.\mathsf{view}^+(t') = p_j.\mathsf{max_views}[k](t') = v' \geq v - 1. \quad (12)$$

Also, for all correct processes p_l, $p_j.\mathsf{max_views}[l](t') < v + 1$: otherwise, some correct process sent $\mathtt{WISH}(v'')$ with $v'' \geq v + 1$ at $t'' < t'$, and therefore, by (11), some correct process attempted to advance from a view $\geq v$ earlier than t', which is impossible. Thus,

$$p_j.\mathsf{view}(t') = p_j.\mathsf{view}^+(t') = p_j.\mathsf{max_views}[k](t') < v + 1.$$

Together with (12), this implies

$$p_j.\mathsf{view}(t') = p_j.\mathsf{view}^+(t') = v.$$

Hence, $\mathsf{LV}_j(t') = v$, and therefore, p_j attempts to advance from v at t'. Thus, $v' = v$ and $t' \leq t$, as required. □

Lemma 14 *Validity holds:* $\forall i, v.\, E_i(v + 1)\downarrow \implies A_{\text{first}}(v) \downarrow \wedge A_{\text{first}}(v) < E_i(v + 1)$.

Proof Since p_i enters a view $v+1$, we have $p_i.\mathsf{view}(E_i(v+1)) = p_i.\mathsf{view}^+(E_i(v + 1)) = v + 1$. By the definitions of view and view$^+$, $v + 1$ is both the lowest view among the highest $2f + 1$ views in $p_i.\mathsf{max_views}(E_i(v + 1))$, and the lowest view among the highest $f + 1$ views in $p_i.\mathsf{max_views}(E_i(v+1))$. Hence, $p_i.\mathsf{max_views}(E_i(v+1))$ includes $f + 1$ entries equal to $v + 1$. Then there exists a time $t' < E_i(v + 1)$ at which some correct process sends $\mathtt{WISH}(v + 1)$. Hence, by Lemma 13, there exists a time $t \leq t' < E_i(v + 1)$ at which some correct process attempts to advance from v. □

In the following we use the next technical lemma, following from Lemmas 13 and 14.

Lemma 15 *For all times t and views $v > 0$, if a correct process sends $\mathtt{WISH}(v)$ at t, then there exists a time $t' \leq t$ such that some correct process attempts to advance from view 0 at t'.*

Proof Consider the earliest time $t_k \leq t$ at which some correct process p_k sends $\mathtt{WISH}(v_k)$ for some view v_k. By Lemma 13, there exists a time $t_j \leq t_k$ at which some correct process attempts to advance from $v_k - 1 \geq 0$, and therefore, sends $\mathtt{WISH}(v_k)$ at t_j. Since t_k is the earliest time when this could happen, we have $t_j = t_k$. Also, if $v_k - 1 > 0$, then $E_k(v_k - 1)$ is defined, and hence, by Lemma 14, some correct process attempts to advance from $v_k - 2$ by sending $\mathtt{WISH}(v_k - 1)$ earlier than $t_j = t_k$, which cannot happen. Thus, $v_k = 1$ and at t_k, p_k attempts to advance from 0, as required. □

Lemma 16 below establishes that the views sent in the WISH messages by the same process can only increase. Its proof relies on the next proposition, stating some simple invariants that follow immediately from the structure of the code.

Proposition 8 *Let p_i be a correct process. Then:*

1. $\forall v. \forall t.\ p_i$ *sends* $\mathtt{WISH}(v)$ at $t \implies v \in \{p_i.\mathsf{view}^+(t), p_i.\mathsf{view}^+(t) + 1\}$.
2. $\forall v. \forall t.\ p_i$ *sends* $\mathtt{WISH}(v)$ at $t \wedge v = p_i.\mathsf{view}^+(t) + 1 \implies$ $p_i.\mathsf{view}^+(t) = p_i.\mathsf{view}(t) \wedge p_i.\mathsf{advanced}(t) = \text{TRUE}$.

Lemma 16 *For all views $v, v' > 0$, if a correct process sends $\mathtt{WISH}(v)$ before sending $\mathtt{WISH}(v')$, then $v \leq v'$.*

Proof Let s and s' such that $s < s'$ be the times at which a correct process p_i sends $\mathtt{WISH}(v)$ and $\mathtt{WISH}(v')$ messages, respectively. We show that $v' \geq v$. By Proposition 8(1), $v \in \{p_i.\mathsf{view}^+(s), p_i.\mathsf{view}^+(s) + 1\}$ and $v' \in \{p_i.\mathsf{view}^+(s'), p_i.\mathsf{view}^+(s') + 1\}$. Hence, if $v = p_i.\mathsf{view}^+(s)$ or $v' = p_i.\mathsf{view}^+(s') + 1$, then we get $v \leq v'$ from the fact that $p_i.\mathsf{view}^+$ is non-decreasing. It thus remains to consider the case when $v = p_i.\mathsf{view}^+(s) + 1$ and $v' = p_i.\mathsf{view}^+(s')$. In this case by Proposition 8(2), $p_i.\mathsf{view}^+(s) = p_i.\mathsf{view}(s)$ and $p_i.\mathsf{advanced}(s) = \text{TRUE}$. We now consider several cases depending on the line at which $\mathtt{WISH}(v')$ is sent.

– $\mathtt{WISH}(v')$ is sent at lines 2 or 6. Then $v' = p_i.\mathsf{view}^+(s') = \max(p_i.\mathsf{view}(s') + 1, p_i.\mathsf{view}^+(s'))$. Since $p_i.\mathsf{view}$ is non-decreasing, we get $p_i.\mathsf{view}^+(s') \geq p_i.\mathsf{view}(s') + 1 > p_i.\mathsf{view}(s') \geq p_i.\mathsf{view}(s) = p_i.\mathsf{view}^+(s)$. Hence, $p_i.\mathsf{view}^+(s') > p_i.\mathsf{view}^+(s)$, and therefore, $v' = p_i.\mathsf{view}^+(s') \geq p_i.\mathsf{view}^+(s) + 1 = v$, as required.

– $\mathtt{WISH}(v')$ is sent at line 8. Then $p_i.\mathsf{advanced}(s') = \text{FALSE}$. Since $p_i.\mathsf{advanced}(s) = \text{TRUE}$, there exists a time s'' such that $s < s'' < s'$ and p_i enters a view at s''. By the view entry condition $p_i.\mathsf{view}(s'') > p_i.prev_v(s'')$. Since $p_i.\mathsf{view}$ is non-decreasing, we get $p_i.\mathsf{view}^+(s') \geq p_i.\mathsf{view}(s') \geq p_i.\mathsf{view}(s'') > p_i.\mathsf{view}(s) = p_i.\mathsf{view}^+(s)$. Thus, $p_i.\mathsf{view}^+(s') > p_i.\mathsf{view}^+(s)$ and therefore, $v' = p_i.\mathsf{view}^+(s') \geq p_i.\mathsf{view}^+(s) + 1 = v$, as required.
– $\mathtt{WISH}(v')$ is sent at line 18. Then $p_i.\mathsf{view}^+(s') > p_i.prev_v^+(s') \geq p_i.\mathsf{view}^+(s)$, and therefore, $v' = p_i.\mathsf{view}^+(s') \geq p_i.\mathsf{view}^+(s) + 1 = v$, as required.

□

In order to cope with message loss before GST, every correct process retransmits the highest WISH it sent every ρ time units, according to its local clock (lines 4-8). Eventually, one of these retransmissions will occur after GST, and therefore, there exists a time by which all correct processes are guaranteed to send their highest WISHes at least once after GST. The earliest such time, $\overline{\mathsf{GST}}$, is defined as follows:

$$\overline{\mathsf{GST}} = \begin{cases} \mathsf{GST} + \rho, & \text{if } A_{\text{first}}(0) < \mathsf{GST}; \\ A_{\text{first}}(0), & \text{otherwise.} \end{cases}$$

From this definition it follows that

$$\overline{\mathsf{GST}} \geq \mathsf{GST}. \tag{13}$$

The following lemma formalizes the key property of $\overline{\mathsf{GST}}$.

Lemma 17 *For all correct processes p_i, times $t \geq \overline{\mathsf{GST}}$, and views v, if p_i sends $\mathtt{WISH}(v)$ at a time $\leq t$, then there exists a view $v' \geq v$ and a time t' such that $\mathsf{GST} \leq t' \leq t$ and p_i sends $\mathtt{WISH}(v')$ at t'.*

Proof Let $s \leq t$ be the time at which p_i sends $\mathtt{WISH}(v)$. We consider two cases. Suppose first that $A_{\text{first}}(0) \geq \mathsf{GST}$. By Lemma 15, $s \geq A_{\text{first}}(0)$, and therefore, $\mathsf{GST} \leq s \leq t$. Thus, choosing $t' = s$ and $v' = v$ validates the lemma. Suppose next that $A_{\text{first}}(0) < \mathsf{GST}$. Then by the definition of $\overline{\mathsf{GST}}$, $t \geq \mathsf{GST} + \rho$. If $s \geq \mathsf{GST}$, then $\mathsf{GST} \leq s \leq t$, and therefore, choosing $t' = s$ and $v' = v$ validates the lemma. Assume now that $s < \mathsf{GST}$. Since after GST the p_i's local clock advances at the same rate as real time, there exists a time s' satisfying $\mathsf{GST} \leq s' \leq t$ such that p_i executes the periodic retransmission code in lines 4-8 at s'. We now show that

$$p_i.\mathsf{advanced}(s') \vee p_i.\mathsf{view}^+(s') > 0. \tag{14}$$

Since p_i already sent a WISH message at $s < \mathsf{GST} \leq s'$, by the structure of the code,

$$p_i.\mathsf{advanced}(s) \vee p_i.\mathsf{view}^+(s) > 0.$$

If $p_i.\mathsf{view}^+(s) > 0$, then since $p_i.\mathsf{view}^+$ is non-decreasing, $p_i.\mathsf{view}^+(s') > 0$, and therefore, (14) holds. Assume now that $p_i.\mathsf{advanced}(s)$. If $p_i.\mathsf{advanced}(s')$, then (14) holds too. We therefore consider the case when $\neg p_i.\mathsf{advanced}(s')$. Then there exists a time $s \leq s'' \leq s'$ at which p_i enters the view $p_i.\mathsf{view}(s'') > 0$. Hence, $p_i.\mathsf{view}^+(s') \geq p_i.\mathsf{view}^+(s'') \geq p_i.\mathsf{view}(s'') > 0$, validating (14). Thus, (14) holds in all cases. Therefore, at s' the process p_i sends $\mathtt{WISH}(v')$ for some view v'. By Lemma 16, $v' \geq v$, and above we established $\mathsf{GST} \leq s' \leq t$, as required. □

The following lemma is used to prove Bounded Entry.

Lemma 18 *Consider a view $v > 0$ and assume that v is entered by a correct process. If $E_{\text{first}}(v) \geq \overline{\mathsf{GST}}$, and no correct process attempts to advance from v before $E_{\text{first}}(v)+2\delta$, then all correct processes enter v and $E_{\text{last}}(v) \leq E_{\text{first}}(v) + 2\delta$.*

Proof If some correct process attempts to advance from a view $v' > v$ before $E_{\text{first}}(v)+2\delta$, then by Proposition 1, some correct process must also enter the view $v+1$. By Lemma 14, this implies that some correct process attempts to advance from v before $E_{\text{first}}(v) + 2\delta$, contradicting the lemma's premise. Thus, no correct process attempts to advance from any view $v' \geq v$ before $E_{\text{first}}(v) + 2\delta$, and therefore, by Lemma 13, no correct process can send $\mathtt{WISH}(v')$ with $v' > v$ earlier than $E_{\text{first}}(v) + 2\delta$. Once any such $\mathtt{WISH}(v')$ is sent, it will take a non-zero time until it is received by any correct process. Thus, we have:

(*) no correct process receives $\mathtt{WISH}(v')$ with $v' > v$ from a correct process until after $E_{\text{first}}(v) + 2\delta$.

Let p_i be a correct process that enters v at $E_{\text{first}}(v)$. By the view entry condition, $p_i.\mathsf{view}(E_{\text{first}}(v)) = v$, and therefore $p_i.\mathsf{max_views}(E_{\text{first}}(v))$ includes $2f + 1$ entries $\geq v$. At least $f + 1$ of these entries belong to correct processes, and by (*), none of them can be $> v$. Hence, there exists a set C of $f + 1$ correct processes, each of which sends $\mathtt{WISH}(v)$ to all processes before $E_{\text{first}}(v)$.

Since $E_{\text{first}}(v) \geq \overline{\mathsf{GST}}$, by Lemma 17, every $p_j \in C$ also sends $\mathtt{WISH}(v')$ with $v' \geq v$ at some time s_j such that $\mathsf{GST} \leq s_j \leq E_{\text{first}}(v)$. Then by (*) we have $v' = v$. It follows that each $p_j \in C$ is guaranteed to send $\mathtt{WISH}(v)$ to all correct processes between GST and $E_{\text{first}}(v)$. Since all messages sent by correct processes after GST are guaranteed to be received by all correct processes within δ of their transmission, by

$E_{\text{first}}(v)+\delta$ all correct processes will receive $\mathtt{WISH}(v)$ from at least $f+1$ correct processes.

Consider an arbitrary correct process p_j and let $t_j \leq E_{\text{first}}(v)+\delta$ be the earliest time by which p_j receives $\mathtt{WISH}(v)$ from $f+1$ correct processes. By (*), no correct process sends $\mathtt{WISH}(v')$ with $v' > v$ before $t_j < E_{\text{first}}(v)+2\delta$. Thus, $p_j.\mathsf{max_views}(t_j)$ includes at least $f+1$ entries equal to v and at most f entries $> v$, so that $p_j.\mathsf{view}^+(t_j) = v$. Then p_j sends $\mathtt{WISH}(v)$ to all processes no later than $t_j \leq E_{\text{first}}(v)+\delta$. Since $E_{\text{first}}(v) \geq \overline{\mathsf{GST}}$, by Lemma 17, p_j also sends $\mathtt{WISH}(v')$ with $v' \geq v$ in-between GST and $E_{\text{first}}(v)+\delta$. By (*), $v'=v$, and therefore, p_j must have sent $\mathtt{WISH}(v)$ to all processes sometime between GST and $E_{\text{first}}(v)+\delta$. Hence, all correct processes are guaranteed to send $\mathtt{WISH}(v)$ to all correct processes between GST and $E_{\text{first}}(v)+\delta$.

Consider an arbitrary correct process p_k and let $t_k \leq E_{\text{first}}(v)+2\delta$ be the earliest time by which p_k receives $\mathtt{WISH}(v)$ from all correct processes. Then by (*), all entries of correct processes in $p_k.\mathsf{max_views}(t_k)$ are equal to v. Since there are at least $2f+1$ correct processes: *(i)* at least $2f+1$ entries in $p_k.\mathsf{max_views}(t_k)$ are equal to v, and *(ii)* one of the $f+1$ highest entries in $p_k.\mathsf{max_views}(t_k)$ is equal to v. From *(i)*, $p_k.\mathsf{view}^+(t_k) \geq p_k.\mathsf{view}(t_k) \geq v$, and from *(ii)*, $p_k.\mathsf{view}(t_k) \leq p_k.\mathsf{view}^+(t_k) \leq v$. Therefore, $p_k.\mathsf{view}(t_k) = p_k.\mathsf{view}^+(t_k) = v$, so that p_k enters v no later than $t_k \leq E_{\text{first}}(v)+2\delta$. We have thus shown that by $E_{\text{first}}(v)+2\delta$, all correct processes enter v, as required. □

Lemma 19 *Bounded Entry holds for $d=2\delta$ and the $\mathcal{V}$ defined in Theorem 3. Namely, consider a view $v \geq \mathcal{V}$ and assume that v is entered by a correct process. If no correct process attempts to advance from v before $E_{\text{first}}(v)+2\delta$, then all correct processes enter v and $E_{\text{last}}(v) \leq E_{\text{first}}(v)+2\delta$.*

Proof Consider the $\mathcal{V}$ defined in Theorem 3. It is easy to see that

$$\mathcal{V} = \max\{v \mid (E_{\text{first}}(v)\downarrow \wedge E_{\text{first}}(v) < \overline{\mathsf{GST}}) \vee v=0\}+1. \tag{15}$$

Then $\forall v \geq \mathcal{V}.\, E_{\text{first}}(v)\downarrow \implies E_{\text{first}}(v) \geq \overline{\mathsf{GST}}$. Bounded Entry now follows from Lemma 18. □

Lemma 20 *Startup holds: suppose there exists a set P of $f+1$ correct processes such that $\forall p_i \in P.\, A_i(0)\downarrow$; then eventually some correct process enters view 1.*

Proof Assume by contradiction that there exists a set P of $f+1$ correct processes such that $\forall p_i \in P.\, A_i(0)\downarrow$, and no correct process enters the view 1. By Proposition 1, the latter implies

$$\forall v' > 0.\, E_{\text{first}}(v')\uparrow. \tag{16}$$

Then by Lemma 13 we have

$$\forall t.\, \forall v' > 1.\, \forall p_i.\, \neg(p_i \text{ sends } \mathtt{WISH}(v') \text{ at } t \wedge p_i \text{ is correct}). \tag{17}$$

Let $T_1 = \max(\overline{\mathsf{GST}}, A_{\text{last}}(0))$. Since there exists a set P of $f+1$ correct processes that attempt to advance from view 0, each $p_i \in P$ sends $\mathtt{WISH}(v_i)$ with $v_i > 0$ before T_1. Since $T_1 \geq \overline{\mathsf{GST}}$, by Lemma 17, there exists a view $v_i' \geq 1$ and a time s_i such that $\mathsf{GST} \leq s_i \leq T_1$ and p_i sends $\mathtt{WISH}(v_i')$ at s_i. By (17), $v_i'=1$. Since the links are reliable after GST, the $\mathtt{WISH}(1)$ sent by p_i at s_i will be received by all correct processes.

Thus, there exists a time $T_2 \geq T_1 \geq \overline{\mathsf{GST}}$ by which all correct processes have received $\mathtt{WISH}(1)$ from all processes in P. Fix an arbitrary correct process p_j. Since all process in P are correct, all entries in $p_j.\mathsf{max_views}(T_2)$ associated with the processes in P are equal to 1. Since $|P|=f+1$, $p_j.\mathsf{max_views}(T_2)$ includes at least $f+1$ entries ≥ 1, and therefore, $p_j.\mathsf{view}^+(T_2) \geq 1$. Hence, p_j sends $\mathtt{WISH}(v_j)$ with $v_j \geq 1$ no later than T_2. Since $T_2 \geq \overline{\mathsf{GST}}$, by Lemma 17 there exists a view $v_j' \geq 1$ and a time s_j such that $\mathsf{GST} \leq s_j \leq T_2$ and p_j sends $\mathtt{WISH}(v_j')$ at s_j. By (17), $v_j'=1$. Since the links are reliable after GST, the $\mathtt{WISH}(1)$ sent by p_j will be received by all correct processes.

Thus, there exists a time $T_3 \geq T_2 \geq \overline{\mathsf{GST}}$ by which all correct processes have received $\mathtt{WISH}(1)$ from all correct processes. Fix an arbitrary correct process p_k. By (17), all entries of correct processes in $p_k.\mathsf{max_views}(T_3)$ are equal to 1. Since there are at least $2f+1$ correct processes: *(i)* at least $2f+1$ entries in $p_k.\mathsf{max_views}(T_3)$ are equal to 1, and *(ii)* one of the $f+1$ highest entries in $p_k.\mathsf{max_views}(T_2)$ is equal to 1. From *(i)*, $p_k.\mathsf{view}^+(T_2) \geq p_k.\mathsf{view}(T_2) \geq 1$, and from *(ii)*, $p_k.\mathsf{view}(T_2) \leq p_k.\mathsf{view}^+(T_2) \leq 1$. Hence, $p_k.\mathsf{view}(T_2) = p_k.\mathsf{view}^+(T_2) = 1$, and therefore, p_k enters view 1 by T_2, contradicting (16). □

Lemma 21 *Progress holds: consider a view $v > 0$ that is entered by a correct process, and suppose there exists a set P of $f+1$ correct processes such that*

$$\forall p_i \in P.\, E_i(v)\downarrow \implies A_i(v)\downarrow; \tag{18}$$

then eventually some correct process enters $v+1$.

Proof Assume by contradiction that the required does not hold. Then, there exists a view $v > 0$ such that some correct process enters v, (18) holds, and no correct process enters the view $v+1$. By Proposition 1, the latter implies that

$$\forall v' > v.\, E_{\text{first}}(v')\uparrow. \tag{19}$$

Thus, by Lemma 13, we have

$$\forall t.\forall v' > v+1.\forall p_i.\neg(p_i \text{ sends } \texttt{WISH}(v') \text{ at } t \wedge p_i \text{ is correct}). \quad (20)$$

Let $T_1 = \max(\overline{\mathsf{GST}}, E_{\text{first}}(v))$. Since some correct process entered v by T_1, there exists a set C consisting of $f+1$ correct processes all of which sent $\texttt{WISH}(v')$ with $v' \geq v$ before T_1. Consider $p_i \in C$ and let $t_i \leq T_1$ be a time such that at t_i the process p_i sends $\texttt{WISH}(v_i)$ with $v_i \geq v$. Since $T_1 \geq \overline{\mathsf{GST}}$, by Lemma 17, there exists a view $v_i' \geq v_i$ and a time s_i such that $\mathsf{GST} \leq s_i \leq T_1$ and p_i sends $\texttt{WISH}(v_i')$ at s_i. By (20), we have $v_i' \in \{v, v+1\}$. Since the links are reliable after GST, the $\texttt{WISH}(v_i')$ sent by p_i at s_i will be received by all correct processes.

Thus, there exists a time $T_2 \geq T_1 \geq \overline{\mathsf{GST}}$ by which all correct processes have received $\texttt{WISH}(v')$ with $v' \in \{v, v+1\}$ from all processes in C. Consider an arbitrary correct process p_j. By (20), the entry of every process in C in $p_j.\mathsf{max_views}(T_2)$ is equal to either v or $v+1$. Since $|C| \geq f+1$ and all processes in C are correct, $p_j.\mathsf{max_views}(T_2)$ includes at least $f+1$ entries $\geq v$. Thus, $p_j.\mathsf{view}^+(T_2) \geq v$, and therefore, p_j sends $\texttt{WISH}(v_j)$ with $v_j \geq v$ no later than at T_2. By (20), $v_j \in \{v, v+1\}$. Since $T_2 \geq \overline{\mathsf{GST}}$, by Lemma 17, there exists a view $v_j' \geq v$ and a time s_j such that $\mathsf{GST} \leq s_j \leq t_j$ and p_j sends $\texttt{WISH}(v_j')$ at s_j. By (20), $v_j' \in \{v, v+1\}$. Since the links are reliable after GST, the $\texttt{WISH}(v_j')$ sent by p_j at s_j will be received by all correct processes.

Thus, there exists a time $T_3 \geq T_2 \geq \overline{\mathsf{GST}}$ by which all correct processes have received $\texttt{WISH}(v')$ such that $v' \in \{v, v+1\}$ from all correct processes. Consider an arbitrary correct process p_k, and suppose that p_k is a member of the set P stipulated by the lemma's premise. Then at T_3, all entries of correct processes in $p_k.\mathsf{max_views}$ are $\geq v$. By (20), each of these entries is equal to either v or $v+1$. Since at least $2f+1$ processes are correct: *(i)* at least $2f+1$ entries in $p_k.\mathsf{max_views}(T_3)$ are $\geq v$, and *(ii)* one of the $f+1$ highest entries in $p_k.\mathsf{max_views}(T_3)$ is $\leq v+1$. From *(i)*, $p_k.\mathsf{view}^+(T_3) \geq p_k.\mathsf{view}(T_3) \geq v$, and from *(ii)*, $p_k.\mathsf{view}(T_3) \leq p_k.\mathsf{view}^+(T_3) \leq v+1$. Hence, $p_k.\mathsf{view}(T_3), p_k.\mathsf{view}^+(T_3) \in \{v, v+1\}$. Since no correct process enters $v+1$, $p_k.\mathsf{view}(T_3)$ and $p_k.\mathsf{view}^+(T_3)$ cannot be both simultaneously equal to $v+1$. Thus, $p_k.\mathsf{view}(T_3) = v$, and either $p_k.\mathsf{view}^+(T_3) = v$ or $p_k.\mathsf{view}^+(T_3) = v+1$. If $p_k.\mathsf{view}^+(T_3) = v+1$, then p_k has sent $\texttt{WISH}(v_k)$ with $v_k = v+1$ when $p_k.\mathsf{view}^+$ has first become equal to $v+1$ sometime before T_3. On the other hand, if $p_k.\mathsf{view}(T_3) = p_k.\mathsf{view}^+(T_3) = v$, then p_k has entered v at some time $t \leq T_3$. Since $p_k \in P$, by (18), there exists a time $t' \geq t$ such that p_k attempts to advance from v at t', and therefore, sends $\texttt{WISH}(v_k)$ with $v_k \geq v+1$ at t'. By (20), $v_k \leq v+1$, and therefore, $v_k = v+1$. Thus, there exists a time $t_k \geq T_3$ by which p_k sends $\texttt{WISH}(v+1)$ to all processes. Since $t_k \geq T_3 \geq \overline{\mathsf{GST}}$, by Lemma 17, there exists a view $v_k' \geq v+1$ and a time s_k such that $\mathsf{GST} \leq s_k \leq t_k$ and p_k sends $\texttt{WISH}(v_k')$ at s_k. By (20), $v_k' = v+1$. Since the links are reliable after GST, the $\texttt{WISH}(v+1)$ sent by p_k will be received by all correct processes.

Thus, there exists a time $T_4 \geq T_3 \geq \overline{\mathsf{GST}}$ by which all correct processes have received $\texttt{WISH}(v+1)$ from all processes in P. Fix an arbitrary correct process p_l. Since all process in P are correct, by (20), all entries in $p_l.\mathsf{max_views}(T_4)$ associated with the processes in P are equal to $v+1$. Since $|P| = f+1$, $p_l.\mathsf{max_views}(T_4)$ includes at least $f+1$ entries equal to $v+1$, and therefore, $p_l.\mathsf{view}^+(T_4) \geq v+1$. Hence, p_l sends $\texttt{WISH}(v_l)$ with $v_l \geq v+1$ no later than T_4. Since $T_4 \geq \overline{\mathsf{GST}}$, by Lemma 17 there exists a view $v_l' \geq v+1$ and a time s_l such that $\mathsf{GST} \leq s_l \leq T_4$ and p_l sends $\texttt{WISH}(v_l')$ at s_l. By (20), $v_l' = v+1$. Since the links are reliable after GST, the $\texttt{WISH}(v+1)$ sent by p_l will be received by all correct processes.

Thus, there exists a time $T_5 \geq T_4 \geq \overline{\mathsf{GST}}$ by which all correct processes have received $\texttt{WISH}(v+1)$ from all correct processes. Fix an arbitrary correct process p_m. By (20), all entries of correct processes in $p_m.\mathsf{max_views}(T_5)$ are equal to $v+1$. Since there are at least $2f+1$ correct processes: *(i)* at least $2f+1$ entries in $p_m.\mathsf{max_views}(T_5)$ are equal to $v+1$, and *(ii)* one of the $f+1$ highest entries in $p_m.\mathsf{max_views}(T_5)$ is equal to $v+1$. From *(i)*, $p_m.\mathsf{view}^+(T_5) \geq p_m.\mathsf{view}(T_5) \geq v+1$, and from *(ii)*, $p_m.\mathsf{view}(T_5) \leq p_l.\mathsf{view}^+(T_5) \leq v+1$. Hence, $p_m.\mathsf{view}(T_5) = p_m.\mathsf{view}^+(T_5) = v+1$, and therefore, p_m enters $v+1$ by T_5, contradicting (19). □

Proof of Theorem 1 Monotonicity is satisfied trivially. Validity, Bounded Entry, Startup, and Progress are established by Lemmas 14, 19, 20, and 21, respectively. □

C Proof of the synchronizer performance properties (Theorem 3)

The following lemma bounds the latency of entering v as a function of the time by which all correct processes have sent such $\texttt{WISH}$es.

Lemma 22 *For all views $v > 0$ and times s, if all correct processes p_i send $\texttt{WISH}(v_i)$ with $v_i \geq v$ no later than at s, and some correct process enters v, then $E_{\text{last}}(v) \leq \max(s, \overline{\mathsf{GST}}) + \delta$.*

Proof Fix an arbitrary correct process p_i that sends $\texttt{WISH}(v_i)$ with $v_i \geq v$ to all processes at time $t_i \leq s \leq \max(s, \overline{\mathsf{GST}})$. Since $\max(s, \overline{\mathsf{GST}}) \geq \overline{\mathsf{GST}}$, by Lemma 17 there exists a time t_i' such that $\mathsf{GST} \leq t_i' \leq \max(s, \overline{\mathsf{GST}})$ and at t_i', p_i sends $\texttt{WISH}(v_i')$ with $v_i' \geq v_i \geq v$ to all processes. Since $t_i' \geq \mathsf{GST}$,

all correct processes receive $\mathtt{WISH}(v_i')$ from p_i no later than at $t_i' + \delta \le \max(s, \overline{\mathsf{GST}}) + \delta$.

Consider an arbitrary correct process p_j and let $t_j \le \max(s, \overline{\mathsf{GST}}) + \delta$ be the earliest time by which p_j receives $\mathtt{WISH}(v_i')$ with with $v_i' \ge v$ from each correct processes p_i. Thus, at t_j, the entries of all correct processes in $p_j.\mathsf{max_views}$ are occupied by views $\ge v$. Since at least $2f+1$ entries in $p_j.\mathsf{max_views}$ belong to correct processes, the $(2f+1)$th highest entry is $\ge v$. Thus, $p_j.\mathsf{view}(t_j) \ge v$. Since $p_j.\mathsf{view}$ is non-decreasing, there exists a time $t_j' \le t_j$ at which $p_j.\mathsf{view}$ first became $\ge v$. If $p_j.\mathsf{view}(t_j') = p_j.\mathsf{view}^+(t_j') = v$, then p_j enters v at t_j'. Otherwise, either $p_j.\mathsf{view}(t_j') > v$ or $p_j.\mathsf{view}^+(t_j') > v$. Since both $p_j.\mathsf{view}$ and $p_j.\mathsf{view}^+$ are non-decreasing, p_j will never enter v after t_j'. Thus, a correct process cannot enter v after $\max(s, \overline{\mathsf{GST}}) + \delta$. Since by the lemma's premise, some correct process does enter v, $E_{\text{last}}(v) \le \max(s, \overline{\mathsf{GST}}) + \delta$, as needed. □

The next lemma gives an upper bound on the duration of time a correct process may spend in a view before sending a $\mathtt{WISH}$ for a higher view.

Lemma 23 *Let p_k be a correct process that enters a view v. Then p_k sends $\mathtt{WISH}(v_k)$ with $v_k \ge v+1$ no later than at $T_{\text{last}}(v)$.*

Proof Suppose that p_k enters a view $v > 0$ at time $\mathsf{GST} \le s_k \le E_{\text{last}}(v)$. Then

$$p_k.\mathsf{view}(s_k) = p_k.\mathsf{view}^+(s_k) = v.$$

By the definition of $T_{\text{last}}(v)$, there exists a time s_k' such that

$$s_k \le s_k' \le T_{\text{last}}(v),$$

and at s_k', p_k either attempts to advance from v or enters a view $v' > v$. If p_k attempts to advance from v at s_k', then p_k sends $\mathtt{WISH}(v_k)$ with $v_k = \max(p_k.\mathsf{view}(s_k') + 1, p_k.\mathsf{view}^+(s_k'))$. Since both $p_k.\mathsf{view}$ and $p_k.\mathsf{view}^+$ are non-decreasing, we have $p_k.\mathsf{view}(s_k') \ge v$ and $p_k.\mathsf{view}^+(s_k') \ge v$. Thus, $v_k \ge v+1$, as required. On the other hand, if p_k enters a view $v' > v$ at s_k', then $v' = p_k.\mathsf{view}(s_k') > p_k.\mathsf{view}(s_k) = v$ and therefore, $p_k.\mathsf{view}^+(s_k') \ge p_k.\mathsf{view}(s_k') \ge v+1$. Since $p_k.\mathsf{view}^+$ is non-decreasing and $p_k.\mathsf{view}^+(s_k) = v$, $p_k.\mathsf{view}^+$ must have changed its value from v to $v_k'' \ge v+1$ at some time s_k'' such that $s_k < s_k'' \le s_k'$. Thus, the condition in line 17 holds at s_k'', which means that p_k sends $\mathtt{WISH}(v_k)$ with $v_k \ge v+1$ at s_k''. Thus, in all cases, p_k sends $\mathtt{WISH}(v_k)$ with $v_k \ge v+1$ no later than at $T_{\text{last}}(v)$, as required. □

The next lemma bounds the time by which every correct process either enters a view $v > 0$, or sends a $\mathtt{WISH}$ messages with a view $> v$.

Lemma 24 *Consider a view $v > 0$ such that some correct process enters v. Then, for all times t, if $t \ge \max(E_{\text{first}}(v), \overline{\mathsf{GST}})$, then $E_{\text{last}}(v) \le t + 2\delta$ and for all correct processes p_k, if p_k never enters v, then, by $E_{\text{last}}(v)$, p_k sends $\mathtt{WISH}(v_k)$ with $v_k \ge v+1$ to all processes.*

Proof Since $v > 0$, $E_{\text{first}}(v)\downarrow$, and $t \ge E_{\text{first}}(v)$, there exists a correct process p_l such that p_l entered v and $E_l(v) \le t$. By the view entry condition, $p_l.\mathsf{view}(E_l(v)) = v$, and therefore $p_l.\mathsf{max_views}(E_l(v))$ includes $2f+1$ entries $\ge v$. Since $f+1$ of these entries belong to correct processes, there exists a set C of $f+1$ correct processes p_i, each of which sent $\mathtt{WISH}(v_i)$ with $v_i \ge v$ to all processes before $E_l(v) \le t$. Since $t \ge \overline{\mathsf{GST}}$, by Lemma 17, p_i sends $\mathtt{WISH}(v_i')$ with $v_i' \ge v_i \ge v$ sometime between GST and t. Since after GST every message sent by a correct process is received by all correct processes within δ of its transmission, the above implies that by $t+\delta$ every correct process receives a $\mathtt{WISH}(v_i')$ with $v_i' \ge v$ from each process $p_i \in C$.

Consider an arbitrary correct process p_j and let $t_j \le t+\delta$ be the earliest time by which p_j receives $\mathtt{WISH}(v_i)$ with $v_i \ge v$ from each process $p_i \in C$. Thus, for all processes $p_i \in C$, $p_j.\mathsf{max_views}[i](t_j) \ge v$. Since $|C| = f+1$, the $(f+1)$th highest entry in $p_j.\mathsf{max_views}[i](t_j)$ is $\ge v$, and therefore, $p_j.\mathsf{view}^+(t_j) \ge v$. Then each correct process p_j sends $\mathtt{WISH}(v_j)$ with $v_j \ge v$ to all correct processes no later than $t_j \le t+\delta$. Since $t+\delta > t \ge \overline{\mathsf{GST}}$ and, and some correct process entered v, by Lemma 22,

$$E_{\text{last}}(v) \le t + 2\delta. \tag{21}$$

In addition, by Lemma 17, there exists a time t_j' such that $\mathsf{GST} \le t_j' \le t+\delta$ and p_j sends $\mathtt{WISH}(v_j')$ with $v_j' \ge v_j \ge v$ at t_j'. Since a message sent by a correct process after GST is received by all correct processes within δ of its transmission, all correct processes must have received $\mathtt{WISH}(v_j')$ with $v_j' \ge v$ from each correct process p_j in-between GST and $t+2\delta$.

Suppose that p_k never enters v, and let t_k be the earliest time $\ge \mathsf{GST}$ by which p_k receives $\mathtt{WISH}(v_j')$ from each correct process p_j; we have $t_k \le t+2\delta$. Since $v_j' \ge v$, and there are $2f+1$ correct processes, $p_k.\mathsf{max_views}(t_k)$ includes at least $2f+1$ entries $\ge v$. Thus, $p_k.\mathsf{view}(t_k) \ge v$. Since p_k never enters v, we have either $p_k.\mathsf{view}^+(t_k) \ge p_k.\mathsf{view}(t_k) \ge v+1$ or $p_k.\mathsf{view}(t_k) = v \wedge p_k.\mathsf{view}^+(t_k) \ge v+1$. Thus, $p_k.\mathsf{view}^+(t_k) \ge v+1$ and therefore, p_k sends $\mathtt{WISH}(v_k)$ with $v_k \ge v+1$ by $t_k \le t+2\delta$, which combined with (21) validates the lemma. □

We are now ready to prove the SMR synchronizer performance bounds.

Theorem 7 *The SMR synchronizer in Fig. 3 satisfies Property A.*

Proof Consider a view v such that $E_{\text{first}}(v)\downarrow$, and let $t = \max(E_{\text{first}}(v), \overline{\mathsf{GST}})$. Since $A_{\text{first}}(0) < \mathsf{GST}$, by the definition of $\overline{\mathsf{GST}}$, $\overline{\mathsf{GST}} = \mathsf{GST} + \rho$. Thus, $t = \max(E_{\text{first}}(v), \mathsf{GST} + \rho)$. By Lemma 24, $E_{\text{last}}(v) \leq t + 2\delta = \max(E_{\text{first}}(v), \mathsf{GST} + \rho) + 2\delta$, as needed. □

Theorem 8 *The SMR synchronizer in Fig. 3 satisfies Property B.*

Proof Consider a view $v \geq 0$ such that $E_{\text{first}}(v+1)\downarrow$. If $v = 0$, then since we assume for all correct processes p_i, $T_i(0)\downarrow$, by Lemma 23, all correct processes send $\texttt{WISH}(v')$ with $v' \geq 0$ to all processes no later than at $T_{\text{last}}(0)$. Thus, by Lemma 22, $E_{\text{last}}(1) \leq \max(T_{\text{last}}(0), \overline{\mathsf{GST}}) + \delta$. If $A_{\text{first}}(0) < \mathsf{GST}$, then $\overline{\mathsf{GST}} = \mathsf{GST} + \rho$, and therefore, $E_{\text{last}}(1) \leq \max(T_{\text{last}}(0), \mathsf{GST} + \rho) + \delta$. Otherwise, $\overline{\mathsf{GST}} = A_{\text{first}}(0) \leq T_{\text{last}}(0)$, so that $E_{\text{last}}(1) \leq T_{\text{last}}(0) + \delta$. Thus, the theorem holds for $v = 0$.

Suppose that $v > 0$. Since some correct process enters $v + 1$, by Proposition 1, some correct process enters view v as well. Consider a correct process p_k. If p_k enters v, then by Lemma 24, $E_k(v) \leq \max(E_{\text{first}}(v), \overline{\mathsf{GST}}) + 2\delta$, and therefore, by Lemma 23, p_k sends $\texttt{WISH}(v_k)$ with $v_k \geq v + 1$ no later than at

$$T_{\text{last}}(v) > \max(E_{\text{first}}(v), \overline{\mathsf{GST}}) + 2\delta. \tag{22}$$

On the other hand, if p_k never enters v, then by Lemma 24, p_k sends $\texttt{WISH}(v_k)$ with $v_k \geq v+1$ than at $\max(E_{\text{first}}(v), \overline{\mathsf{GST}}) + 2\delta$. Thus, every correct process p_k sends $\texttt{WISH}(v_k)$ with $v_k \geq v + 1$ no later than

$$\max(T_{\text{last}}(v), \max(E_{\text{first}}(v), \overline{\mathsf{GST}}) + 2\delta),$$

which by (22), implies that all correct processes send a $\texttt{WISH}$ message with a view $\geq v + 1$ no later than $T_{\text{last}}(v)$. Thus, by Lemma 22, we have

$$E_{\text{last}}(v + 1) \leq \max(T_{\text{last}}(v), \overline{\mathsf{GST}}) + \delta. \tag{23}$$

If $A_{\text{first}}(0) < \mathsf{GST}$, then $\overline{\mathsf{GST}} = \mathsf{GST} + \rho$, and therefore, (23) implies that $E_{\text{last}}(v + 1) \leq \max(T_{\text{last}}(v), \mathsf{GST} + \rho) + \delta$, as required. Otherwise, $\overline{\mathsf{GST}} = A_{\text{first}}(v) \leq T_{\text{last}}(v)$, which by (23) implies that $E_{\text{last}}(v + 1) \leq T_{\text{last}}(v) + \delta$, validating the theorem. □

Proof of Theorem 3 Follows from Theorems 7 and 8. □

References

1. Abraham, I., Devadas, S., Dolev, D., Nayak, K., Ren, L.: Synchronous Byzantine agreement with expected $O(1)$ rounds, expected $O(n^2)$ communication, and optimal resilience. In: Conference on Financial Cryptography and Data Security (FC) (2019)
2. Abraham, I., Gueta, G., Malkhi, D., Alvisi, L., Kotla, R., Martin, J.-P.: Revisiting fast practical Byzantine fault tolerance (2017). arXiv:1712.01367
3. Abraham, I., Nayak, K., Ren, L., Xiang, Z.: Good-case latency of Byzantine broadcast: a complete categorization. In: Symposium on Principles of Distributed Computing (PODC) (2021)
4. Aştefănoaei, L., Chambart, P., Del Pozzo, A., Rieutord, T., Tucci-Piergiovanni, S., Zălinescu, E.: Tenderbake-a solution to dynamic repeated consensus for blockchains. In: Symposium on Foundations and Applications of Blockchain (FAB) (2021)
5. Alistarh, D., Gilbert, S., Guerraoui, R., Travers, C.: Generating fast indulgent algorithms. In: International Conference on Distributed Computing and Networking (ICDCN) (2011)
6. Amoussou-Guenou, Y., Del Pozzo, A., Potop-Butucaru, M., Tucci-Piergiovanni, S.: Correctness of Tendermint-core blockchains. In: Conference on Principles of Distributed Systems (OPODIS) (2018)
7. Amoussou-Guenou, Y., Del Pozzo, A., Potop-Butucaru, M., Tucci-Piergiovanni, S.: Dissecting tendermint. In: Conference on Networked Systems (NETYS) (2019)
8. Awerbuch, Baruch: Complexity of network synchronization. J. ACM **32**(4), 804–823 (1985)
9. Bazzi, R.A., Ding, Y.: Non-skipping timestamps for Byzantine data storage systems. In: Symposium on Distributed Computing (DISC) (2004)
10. Berger, C., Reiser, H.P., Bessani, A.: Making reads in BFT state machine replication fast, linearizable, and live. In: Symposium on Reliable Distributed Systems (SRDS) (2021)
11. Bessani, A.N., Sousa, J., Adílio Pelinson, R.M., Alchieri, E.: State machine replication for the masses with BFT-SMART. In: Conference on Dependable Systems and Networks (DSN) (2014)
12. Biely, M., Widder, J., Charron-Bost, B., Gaillard, A., Hutle, M., Schiper, A.: Tolerating corrupted communication. In: Symposium on Principles of Distributed Computing (PODC) (2007)
13. Bracha, G.: Asynchronous Byzantine agreement protocols. Inf. Comput. **75**(2), 130–143 (1987)
14. Bravo, M., Chockler, G., Gotsman, A.: Making Byzantine consensus live. In: Symposium on Distributed Computing (DISC) (2020)
15. Bravo, M., Chockler, G., Gotsman, A.: Liveness and latency of Byzantine state-machine replication (extended version) (2022). arXiv:2202.06679. https://arxiv.org/abs/2202.06679
16. Bravo, M., Chockler, G., Gotsman, A.: Making Byzantine consensus live. Distrib. Comput. **35**(6), 503–532 (2022)
17. Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on BFT consensus (2018). arXiv:1807.04938
18. Cachin, C.: Personal communication (2022)
19. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to Reliable and Secure Distributed Programming, 2nd edn. Springer, Berlin (2011)
20. Cachin, C., Kursawe, K., Petzold, F., Shoup, V.: Secure and efficient asynchronous broadcast protocols. In: International Cryptology Conference (CRYPTO) (2001)
21. Cachin, C., Vukolić, M.: Blockchain consensus protocols in the wild (keynote talk). In: Symposium on Distributed Computing (DISC) (2017)
22. Castro, M.: Practical Byzantine Fault Tolerance. PhD thesis, Massachusetts Institute of Technology (2001)
23. Castro, M., Liskov, B.: Practical Byzantine fault tolerance. In: Symposium on Operating Systems Design and Implementation (OSDI) (1999)
24. Castro, M., Liskov, B.: Practical Byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. **20**(4), 398–461 (2002)
25. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996)

26. Charron-Bost, B., Schiper, A.: The Heard-Of model: computing in distributed systems with benign faults. Distrib. Comput. **22**(1), 49–71 (2009)
27. Civit, P., Dzulfikar, M.A., Gilbert, S., Gramoli, V., Guerraoui, R., Komatovic, J., Vidigueira, M.: Byzantine consensus is $\theta(n^2)$: the Dolev-Reischuk bound is tight even in partial synchrony! In: Symposium on Distributed Computing (DISC) (2022)
28. Clement, A., Wong, E., Alvisi, L., Dahlin, M., Marchetti, M.: Making Byzantine fault tolerant systems tolerate Byzantine faults. In: Symposium on Networked Systems Design and Implementation (NSDI) (2009)
29. DiemBFT v4: state machine replication in the Diem blockchain. https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf
30. Dolev, D., Halpern, J.Y., Simons, B., Strong, R.: Dynamic fault-tolerant clock synchronization. J. ACM **42**(1), 143–185 (1995)
31. Doudou, A., Garbinato, B., Guerraoui, R.: Abstractions for devising Byzantine-resilient state machine replication. In: Symposium on Reliable Distributed Systems (SRDS) (2000)
32. Dragoi, C., Widder, J., Zufferey, D.: Programming at the edge of synchrony. Proc. ACM Program. Lang. **4**(OOPSLA), 1–30 (2020)
33. Dwork, C., Lynch, N.A., Stockmeyer, L.J.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
34. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)
35. Freiling, F.C., Guerraoui, R., Kuznetsov, P.: The failure detector abstraction. ACM Comput. Surv. **43**(2), 9:1-9:40 (2011)
36. Gafni, E.: Round-by-round fault detectors: unifying synchrony and asynchrony. In: Symposium on Principles of Distributed Computing (PODC) (1998)
37. Gilbert, S., Guerraoui, R., Kowalski, D.R.: On the message complexity of indulgent consensus. In: Symposium on Distributed Computing (DISC) (2007)
38. Golan-Gueta, G., Abraham, I., Grossman, S., Malkhi, D., Pinkas, B., Reiter, M.K., Seredinschi, Dragos-Adrian, Tamir, Orr, Tomescu, Alin: SBFT: A scalable and decentralized trust infrastructure. In: Conference on Dependable Systems and Networks (DSN) (2019)
39. Guerraoui, R.: Indulgent algorithms (preliminary version). In: Symposium on Principles of Distributed Computing (PODC) (2000)
40. Guerraoui, R., Raynal, M.: The information structure of indulgent consensus. IEEE Trans. Comput. **53**(4), 453–466 (2004)
41. Haeberlen, A., Kuznetsov, P.: The fault detection problem. In: Conference on Principles of Distributed Systems (OPODIS) (2009)
42. Herzberg, A., Kutten, S.: Fast isolation of arbitrary forwarding faults. In: Symposium on Principles of Distributed Computing (PODC) (1989)
43. Incorrect by construction-CBC Casper isn't live. https://derekhsorensen.com/docs/CBC_Casper_Flaw.pdf
44. Keidar, I., Shraer, A.: Timeliness, failure-detectors, and consensus performance. In: Symposium on Principles of Distributed Computing (PODC) (2006)
45. Kihlstrom, K.P., Moser, L.E., Melliar-Smith, P.M.: Byzantine fault detectors for solving consensus. Comput. J. **46**(1), 16–35 (2003)
46. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: speculative byzantine fault tolerance. ACM Trans. Comput. Syst. **27**(4), 7:1-7:39 (2010)
47. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
48. Malkhi, D., Reiter, M.: Unreliable intrusion detection in distributed computations. In: Workshop on Computer Security Foundations (CSFW) (1997)
49. Mostéfaoui, A., Raynal, M.: Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In: Symposium on Distributed Computing (DISC) (1999)
50. Naor, O., Baudet, M., Malkhi, D., Spiegelman, A.: Cogsworth: Byzantine view synchronization. In: Cryptoeconomics Systems Conference (CES) (2020)
51. Naor, O., Keidar, I.: Expected linear round synchronization: the missing link for linear Byzantine SMR. In: Symposium on Distributed Computing (DISC) (2020)
52. Pass, R., Shi, E.: Hybrid consensus: efficient consensus in the permissionless model. In: Symposium on Distributed Computing (DISC) (2017)
53. Pass, R., Shi, E.: Thunderella: blockchains with optimistic instant confirmation. In: Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT) (2018)
54. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. **22**(4), 299–319 (1990)
55. Simons, B., Welch, J., Lynch, N.: An overview of clock synchronization. In: Fault-Tolerant Distributed Computing (1986)
56. Sousa, J.: Byzantine State Machine Replication for the Masses. PhD thesis, University of Lisbon (2017)
57. Stathakopoulou, C., David, T., Vukolić, M.: Mir-BFT: high-throughput BFT for blockchains (2019). arXiv:1906.05552
58. Veronese, G.S., Correia, M., Bessani, A.N., Lung, L.C.: Spin one's wheels? Byzantine fault tolerance with a spinning primary. In: Symposium on Reliable Distributed Systems (SRDS) (2009)
59. Yin, M., Malkhi, D., Reiter, M.K., Golan-Gueta, G., Abraham, I.: HotStuff: BFT consensus with linearity and responsiveness. In: Symposium on Principles of Distributed Computing (PODC), (2019)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.